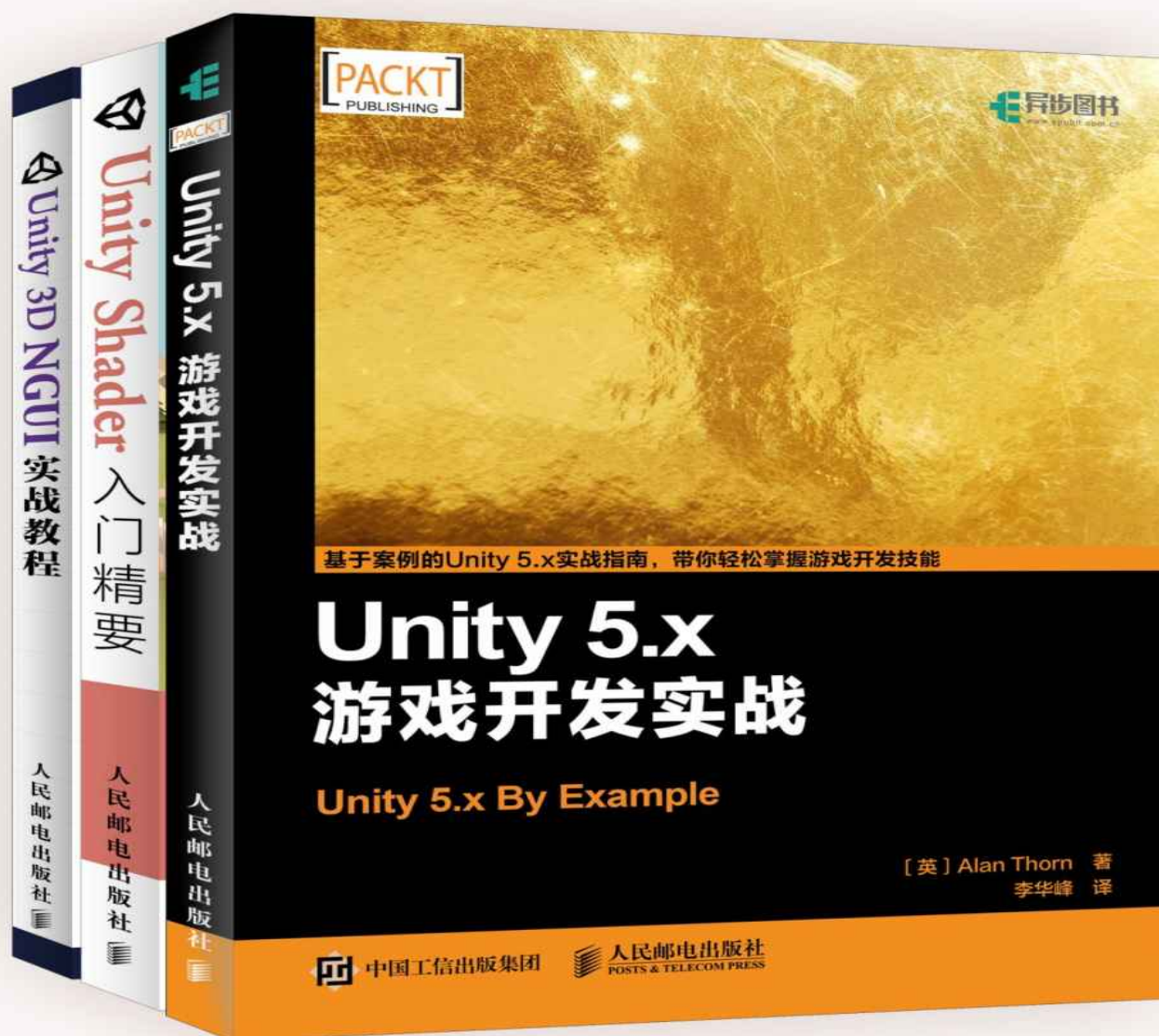


套装共3册

# Unity开发实战 (第1辑)



人民邮电出版社  
POSTS & TELECOM PRESS

异步社区  
人民邮电出版社

配套资源下载请访问 异步社区  
[www.epubit.com.cn](http://www.epubit.com.cn)

# 总 目 录

---

[Unity 5.x游戏开发实战](#)

[Unity Shader入门精要](#)

[Unity 3D NGUI实战教程](#)





基于案例的Unity 5.x实战指南，带你轻松掌握游戏开发技能

# Unity 5.x 游戏开发实战

Unity 5.x By Example

[英] Alan Thorn 著  
李华峰 译



中国工信出版集团



人民邮电出版社  
POSTS & TELECOM PRESS

# 目 录

[版权信息](#)

[版权声明](#)

[内容提要](#)

[作者简介](#)

[审稿人简介](#)

[译者序](#)

[前言](#)

[第1章 金币采集游戏 \(I\)](#)

[1.1 游戏设计](#)

[1.2 从头开始——Unity中的项目](#)

[1.3 项目和项目文件夹](#)

[1.4 开始一个关卡](#)

[1.5 变换和导航](#)

[1.6 场景的建立](#)

[1.7 光源和天空](#)

[1.8 游戏测试与游戏选项卡](#)

[1.9 添加一个水平面](#)

[1.10 添加一个用来采集的金币](#)

[1.11 小结](#)

[第2章 金币采集游戏 \(II\)](#)

[2.1 创建一个金币的材质](#)

[2.2 Unity中的C#脚本](#)

[2.3 对金币进行计数](#)

[2.4 金币采集](#)

[2.5 金币与预设体](#)  
[2.6 定时器和倒计时](#)  
[2.7 庆典和焰火](#)  
[2.8 游戏测试](#)  
[2.9 构建](#)  
[2.10 小结](#)

[第3章 太空射击游戏 \(I\)](#)  
[3.1 对完整的项目进行展望](#)  
[3.2 开始太空射击游戏](#)  
[3.3 创建玩家对象](#)  
[3.4 Player输入](#)  
[3.5 配置游戏中的摄像机](#)  
[3.6 范围的锁定](#)  
[3.7 生命值](#)  
[3.8 死亡和粒子系统](#)  
[3.9 敌人](#)  
[3.10 批量产生敌人](#)  
[3.11 小结](#)

[第4章 太空射击游戏 \(II\)](#)  
[4.1 武器与炮塔](#)  
[4.2 炮弹预设体](#)  
[4.3 炮弹的产生](#)  
[4.4 用户控制](#)  
[4.5 分数和评分——UI和文本对象](#)  
[4.6 计分功能——为文本对象编写脚本](#)  
[4.7 游戏的润色](#)  
[4.8 测试与调试](#)  
[4.9 游戏的构建](#)  
[4.10 小结](#)

[第5章 二维冒险游戏 \(I\)](#)  
[5.1 二维冒险游戏——开始](#)  
[5.2 资源的导入](#)  
[5.3 开始创建游戏的环境](#)  
[5.4 环境物理学](#)

[5.5 创建一个玩家](#)  
[5.6 编写控制玩家移动脚本](#)  
[5.7 优化](#)  
[5.8 小结](#)

[第6章 二维冒险游戏 \(II\)](#)  
[6.1 移动的平台](#)  
[6.2 创建其他的场景——关卡2和关卡3](#)  
[6.3 死亡区域](#)  
[6.4 用户界面中的生命值条](#)  
[6.5 炮弹和伤害](#)  
[6.6 炮塔和炮弹](#)  
[6.7 NPC和任务系统](#)  
[6.8 小结](#)

[第7章 有智慧的敌人 \(I\)](#)  
[7.1 项目概览](#)  
[7.2 入门指南](#)  
[7.3 地形的构建](#)  
[7.4 导航与导航网格](#)  
[7.5 构建一个NPC](#)  
[7.6 创建巡逻的NPC](#)  
[7.7 小结](#)

[第8章 有智慧的敌人 \(II\)](#)  
[8.1 敌人的人工智能——视野范围](#)  
[8.2 有限状态机概述](#)  
[8.3 巡逻状态](#)  
[8.4 追逐状态](#)  
[8.5 攻击状态](#)  
[8.6 小结](#)

[欢迎来到异步社区！](#)

[返回总目录](#)

## 版权信息

书名：Unity 5.x游戏开发实战

ISBN：978-7-115-45598-7

本书由人民邮电出版社发行数字版。版权所有，侵权必究。

---

您购买的人民邮电出版社电子书仅供您个人使用，未经授权，不得以任何方式复制和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

---

• 著 [英] Alan Thorn

译 李华峰

责任编辑 胡俊英

• 人民邮电出版社出版发行 北京市丰台区成寿寺路11号

邮编 100164 电子邮件 315@ptpress.com.cn



网址 <http://www.ptpress.com.cn>

- 读者服务热线: (010)81055410

反盗版热线: (010)81055315

## 版权声明

Copyright ©2016 Packt Publishing. First published in the English language under the title

*Unity 5.x By Example.*

All rights reserved.

本书由英国Packt Publishing公司授权人民邮电出版社出版。未经出版者书面许可，对本书的任何部分不得以任何方式或任何手段复制和传播。

版权所有，侵权必究。

## 内容提要

Unity是一个可以轻松创建各类型互动内容的多平台综合型游戏开发工具，是一个全面整合的专业游戏引擎。本书基于Unity 5.0及以上版本进行讲解，引导读者深度认识并掌握这一重要的游戏开发工具。

本书共分8章，通过4个典型的 game 项目来引导读者进行学习，每两章完成一个 game 案例，案例式的讲解模式更有利于读者快速提升实践能力。金币采集 game 开启了Unity开发之旅，随后的太空射击 game 进一步丰富了各类 game 设计技巧，之后又通过二维冒险 game 完整地呈现了Unity的强大功能，最后通过一个人工智能项目完整地将地形构建、导航等功能有机地整合到 game 当中。

本书几乎包含了学习Unity所需的所有内容，案例式的学习更有助于读者快速掌握开发技巧。本书非常适合那些没有Unity和 game 开发经验的读者，通过阅读本书，读者将掌握使用Unity进行 game 开发的核心技巧。如果读者对 game 开发和Unity本身有着浓厚的兴趣，那将对其学习提供无限助力，学习效果会更加出色。

## 作者简介

**Alan Thorn**是一位屡获殊荣的作家、数学家，同时也是一位独立的视频游戏开发商，目前居住于英国伦敦。他创立了“**Wax Lyrical Games**”游戏开发工作室，并开发了一款广受好评的PC冒险游戏《维塔德男爵：灭绝的复仇》。Alan作为一名自由职业者，曾经服务于世界上一些大型的游戏公司。他曾经在欧洲最著名的机构讲授游戏开发，编写了9本游戏编程方面的图书，这其中就包括十分受欢迎的*Teach Yourself Games Programming*、*Game Engine Design and Implementation*和*UDK Game Development*。Alan还对计算、数学、制图学和哲学很感兴趣。关于他的“**Wax Lyrical Games**”公司的更多信息可以访问<http://www.waxlyricalgames.com>获取。

## 审稿人简介

**Francesco Sapio**毕业于意大利的罗马大学，在校期间获得了优异的成绩，现在正在攻读人工智能和机器人学的硕士学位。

他是一位Unity 3D的专家，同时也是一位熟练的游戏软件开发者，也是一位经验丰富的图形程序使用者。

最近，他刚刚编写了Unity UI Cookbook一书，专门教授读者如何使用Unity开发引人注目并且实用的图形用户界面。此外，他还是Unity Game Development Scripting一书的审校专家。Francesco还是一位音乐家和作曲家，尤其擅长为小电影和视频游戏配乐。近年来，他还从事了演员和舞蹈家的工作。目前，他还是罗马布兰卡乔剧院的特邀嘉宾。

此外，他还是一个非常活泼的人，曾经在罗马的意大利文化中心担任儿童娱乐志愿者。他还为高中和大学的学生讲授数学和音乐的私人课程。

Francesco热爱数学、哲学、逻辑学和破解难题，尤其是开发视频游戏，这一切都源自他对游戏设计和编程开发的热情。读者可以在领英上查看关于他的个人信息。

在这里我要深深感谢我的父母，是他们永无止尽的耐心、热情和支持一直伴随我到现在。而且我也要感谢家庭的所有成员，尤其是我的祖父



母，在我的生命中，他们总是不断地用拉丁语“Ad Maiora”和“Per aspera ad astra”来鼓励我去将事情做得更好。

最后，我要对身边每一个我爱的人表示最大的谢意，尤其是我的女友，我非常感谢她为我所做的一切！

## 译者序

游戏是从现实通往梦境的一个通道，在我们每个人的成长过程中，游戏都扮演着一个不可或缺的角色。直到现在我依然忘不了，在上中学时，每当放学就偷偷跑到街边的游戏厅和同学大战《拳皇》的情景。那些一个个操作简单、画面朴实的游戏，却不知道实现了多少人儿时心中的英雄梦。

随着计算机技术的不断发展，一个个令人惊叹的游戏不断出现，从很早的《金庸群侠传》《星际争霸》《暗黑破坏神》一直到现在那些我也无法叫上名字的新游戏。各种新游戏不断出现，新游戏画面越来越精美，情节越来越丰富，这一切都得益于游戏开发引擎的不断进步。

可以毫不夸张地说，游戏开发者其实就是在扮演着“造梦者”的角色。这个职业在绝大多数人的眼中是极为神秘的，他们应该每天在高耸的大楼里面无休止地敲打着键盘，用那些普通人无法理解的代码创造着一个又一个的虚拟世界。

其实这是一个误解，很多游戏就是游戏开发者在家里利用业余时间完成的，有些开发者甚至可能只懂一点编程的知识。怎么样？听到这些是不是觉得很吃惊。其实每个人都可以成为游戏开发者，需要什么？只要有一个Unity 3D就足够了！

Unity 3D是什么呢？严格来说，这是一款游戏开发的引擎。关于它的详细描述，读者可以自行了解。如果要我来描述Unity 3D，只有两个词——简单、强大。如果你有一家公司、一个水平高超的开发团队及一笔不菲的开发基金，那你就可以用Unity 3D开发出一款脍炙人口的游戏。如果你什么都没有，那么凭着对游戏开发的热情，在家里，一个人从头学起，Unity同样可以帮助你开发出一款风靡世界的游戏！

本书的作者Alan Thorn是一位职业游戏开发者，他所开发的游戏《维塔德男爵：灭绝的复仇》受到了很多人的欢迎。难能可贵的是，他在本书编写中详细地分享了使用Unity开发游戏的案例过程。Alan Thorn绝对是一位与众不同的作家，在他的作品中，极少提到技术以外的问题，全书都是详实的案例，毫无赘言，所有案例准确细致，由浅入深。这一点让我在本书的翻译过程中受益匪浅。

感谢人民邮电出版社的编辑胡俊英，在本书编写的这段时间中她始终支持我的写作，她的鼓励和帮助引导我能顺利翻译完成全部书稿。最后感谢我的母亲，是她将我培养成人，并在人生的每一个关键阶段给我提供帮助；感谢我深爱的妻子及我可爱的儿子，感谢你们在我翻译本书的时候，给我无条件的理解和支持。

——李华峰

## 前言

视频游戏作为一种文化现象，在过去半个世纪已经在世界范围内吸引并娱乐了数十亿人。视频游戏作为一种产业和文化，无论对于开发者还是艺术家来说，都是一个令人兴奋的所在。通过游戏，你的意愿、想法和工作可以去影响更多的人，并且以一种前所未有的方式来塑造和改变一代又一代的人。在最近的一段时间里，兴起了一股游戏开发的平民化运动，目标是游戏的开发过程要更简单，游戏本身要更容易、更广泛地被人们所接受，包括开发者在预算有限的情况下，在家里就能从事游戏开发的工作。推动这项运动的倡导者就是Unity引擎，这也是本书的主题。Unity引擎是一个计算机程序，它可以与你现有的资源途径（例如三维建模软件）协同工作，用来编写可以在多平台和设备，包括Windows、Mac、Linux、Android、iOS和Windows Phone上都能正常工作的视频游戏。使用Unity引擎，开发者可以导入现成的资源（例如音乐、贴图、3D模型等），然后将它们组装成一个有机的整体，通过一个全局的游戏逻辑形成一个游戏世界。Unity引擎是一个令人着迷的程序。最新版本的Unity可以免费下载和使用，而且它可以十分有效地和其他程序（例如GIMP和Blender）协同工作。本书着眼于Unity引擎以及如何使用它来开发一个好玩并且有趣的游戏。学习Unity并不需要什么基础，但是必须对编程语言有一定的了解（如JavaScript、ActionScript、C、C++、Java或者C#）。现在我们以章节的形式来看看本书中都涵盖了哪些内容。

## 本书内容

本书将在实践中探讨Unity引擎的使用，通过具体的游戏实例，一步步地带你进行游戏开发。本书共有8章，重点讲解了4个迥然不同的案例，每两章完成一个案例。接下来，我们来看看这些案例的具体内容。

第1章以第一人称视角金币采集游戏开始了我们的Unity开发之旅。如果你此前对Unity完全不了解并且准备开始你的第一个游戏，那么这就是一个绝佳的切入点。

第2章紧接第1章的内容，并完成了这个游戏。这里假设你已经完成了第1章的内容，并且完成了整个的项目，就为下一章的开始做了铺垫。

第3章标志着第二个项目的开始，我们将专注于一个太空射击游戏的开发。在这里，我们创建一个游戏，在游戏中玩家将要面对迎面而来的敌人。

第4章完成了整个的太空射击项目，这一章的内容紧随上一章，并为项目添加最后的加工。

第5章展示了二维游戏和UI功能的风采。在这里，我们将通过开发一个依靠二维物理属性的侧视图游戏来探讨Unity强大的二维功能。

第6章完成上一章开始的二维冒险游戏，为此进行了最后的完善，并根据总体游戏逻辑将各个环节连接在一起。通过这个实例我们可以



很清晰地了解游戏中的各个部分和各个方面是如何有机地形成一个整体的。

第7章着重研究人工智能，创造了一个可以巡逻、追逐并攻击的敌人，而且也实现了这些敌人的导航功能。

第8章结束了上一章的人工智能项目，和本书的其他内容成为一个有机的整体。在这个项目中，我们将会看到如何使用有限状态机来实现一个强大的人工智能，这个人工智能可以被应用在各种情况下。

## 本书要求

本书几乎包含了学习Unity的所有内容。每一章都提供了注重实用性的真实Unity开发案例，并提供了可以下载和使用的配套文件。除了本书和你的学习兴趣之外，你还需要下载一个最新版本的Unity软件。本书在编写的时候，Unity的最新版是5.3.1。这个软件的个人版

（personal edition）可以免费下载和使用，下载地址为<https://unity3d.com/>。除了Unity之外，如果创建道具、人物角色，或者其他的3D资源，可能需要3D建模软件和动画软件，例如3DS Max、Maya或者Blender等，可能还会需要图像编辑软件，例如Photoshop和GIMP等。Blender是一款免费的软件，可以从<http://www.blender.org/>下载。GIMP也是一款免费的软件，可以从<https://www.gimp.org/>下载。

## 本书的目标读者

本书十分适合那些没有Unity和游戏开发经验，但却希望将游戏开发作为一种爱好或者职业的读者。你也许已经有了一些与游戏开发无关的程序编写或者脚本编辑基本知识，例如C、C++、C#、Java、JavaScript、ActionScript、Python或者其他面向对象编程语言的编程能力。此外，你至少应该要有一些基本的关于游戏开发的重要概念。例如，本书假设你已经知道了3D模型、贴图、音频文件、可执行文件等基础性知识。本书会对这些概念进行简单的讲解，但是不会深入。本书将专注于Unity作为软件开发游戏的功能，每一章都是一篇可以深入研究的教程，并提供了一个可以试玩和扩展的游戏。

## 本书约定

本书中会有很多的文本样式，这些样式用来区分不同种类的信息。下面给出了这些样式的一些实例以及它们含义的解释。

代码段将按照如下格式显示：

```
using UnityEngine;
using System.Collections;

public class Coin : MonoBehaviour
{
    // Use this for initialization
    void Start () {}

    // Update is called once per frame
    void Update () {}
}
```

当代码块中的特定部分需要注意时，相关的行或者条目会被设置为粗体：

```
using UnityEngine;
using System.Collections;

public class Coin : MonoBehaviour
{
    // Use this for initialization
    void Start () {
        Debug.Log ("Object Created");
    }

    // Update is called once per frame
    void Update () {

    }
}
```

新名词和重要的词汇，例如“菜单”“对话框”等将会以粗体显示，显示格式如同“你将需要创建一个新项目”。



警告或重要的注释出现在这样的括号里。



注意

提示和技巧出现在这样的括号里。

## 读者反馈

我们欢迎读者的反馈意见。无论你对本书有什么想法，喜欢或者不喜欢哪些内容，都可以告诉我们。这些反馈意见对我们创作出对大家真正有所帮助的作品至关重要。

你可以将一般的反馈以电子邮件的方式发送到 [feedback@packtpub.com](mailto:feedback@packtpub.com)，并在邮件主题中包含书名。

如果你在某一方面很有造诣，并且愿意著书或参与合著，可以参考我们的作者指南[www.packtpub.com/authors](http://www.packtpub.com/authors)。

## 客户支持

现在你已经是Packt图书的尊贵读者了，我们会尽力帮读者充分利用手中的图书。

## 示例代码下载

可以使用账号在<http://www.packtpub.com> 下载本书的样例代码。如果是通过其他途径购买的本书，则可以访问网址<http://www.packtpub.com/support> 进行注册，这些文件会直接发送到你的邮箱中。

可以按照如下步骤下载代码文件。

1. 使用邮箱地址和密码在我们的网站登录或者注册。
2. 将鼠标移动到顶部的“SUPPORT”选项卡上。

3. 单击“Code Downloads & Errata”。
4. 在查找对话框中输入本书的名字。
5. 选中要下载代码文件的书名。
6. 从下拉列表框中选中购买本书的途径。
7. 单击“Code Download”。

当文件下载之后，需要使用如下软件的最新版来对这个文件进行解压。

Windows操作系统下的WinRAR / 7-Zip

Mac操作系统下的Zipeg / iZip / UnRarX

Linux操作系统下的7-Zip / PeaZip

## 下载本书的彩图版

我们同样提供了使用本书的PDF彩图版，彩色的图片可以更有效地帮助读者理解书中的内容，可以从下面的这个地址下载本书的彩图版[https://www.packtpub.com/sites/default/files/downloads/Unity5xByExample\\_ColorImages.pdf](https://www.packtpub.com/sites/default/files/downloads/Unity5xByExample_ColorImages.pdf)。

## 勘误



虽然我们已尽力确保本书内容正确，但错误仍在所难免。如果读者在书中发现任何文字或者代码错误，就请将这些错误提交给我们，以便帮助我们改进本书的后续版本，避免其他读者产生不必要的误解。如果读者发现了错误，请访问[http://www.packtpub.com/ submit-errata](http://www.packtpub.com/submit-errata)，选择相应图书，单击errata submission form链接，然后填写具体的错误信息即可。勘误一经核实，读者的提交将被接受，此勘误将被上传到本公司网站或添加到现有勘误表中。读者可以通过在<http://www.packtpub.com/support> 上选择书名来查看本书所有的勘误表。

## 侵权声明

版权问题是每一个媒体都要面对的问题。Packt非常重视版权的保护。如果读者发现我们的作品在互联网上以任何形式被非法复制，请立即告知我们相关网址或网站名称，以便我们采取措施。

请把可疑盗版材料的链接发到[copyright@packtpub.com](mailto:copyright@packtpub.com)。

非常感谢你帮助我们保护作者的权益。

## 问题

如果你对本书有任何方面的疑问，可以通过[questions@packpub.com](mailto:questions@packpub.com) 联系我们，我们将尽最大的努力解决。

## 第1章 金币采集游戏 (I)

本章将会以一个十分有趣的采集类游戏作为Unity的入门。即使你以前从来都没有使用过Unity这个游戏引擎，也无需有任何的担心。我们的教程将会一步一步地来指导你熟悉Unity。到了下一章结束的时候，你就可以利用所学的内容构造出一个比较简单但是功能相当完善的游戏。这将是一个非常重要的过程，因为在从始至终的开发中，你将会了解到整个游戏的实现流程。

在本章中将会学习到如下内容：

- Unity游戏的设计
- Unity项目和目录
- Unity资源（Asset）的导入和配置
- Unity游戏关卡的设计
- Unity游戏中的对象
- Unity中的结构树（Hierarchies）

### 1.1 游戏设计

现在就来开始游戏的开发之旅吧。首先，这是一个第一人称视角的游戏，玩家必须在规定时间内通过不断地移动去采集所有的金币。如果玩家在时间耗尽时还没有采集到所有金币，则视为玩家失败，游戏结束。如果玩家在时间耗尽之前就将全部的金币采集完毕，则视为玩家胜利。这款游戏的方向控制采用标准“WASD”的模式，按下“W”键向前运动，按下“A”键向左运动，按下“S”键向右运动，按下“D”键向后运动。视角的变换由鼠标控制，当走到金币的位置时就可以完成金币的采集。如图1.1所示，采用Unity编辑器实现游戏行为的设计。开发这个游戏的最大优势就在于，通过整个的开发过程，可以了解到Unity的全部关键特性，同时无需使用任何额外的软件来产生游戏所需的资源（Asset），例如贴图（Textures）、网格（Meshes）以及材质（Materials）等。

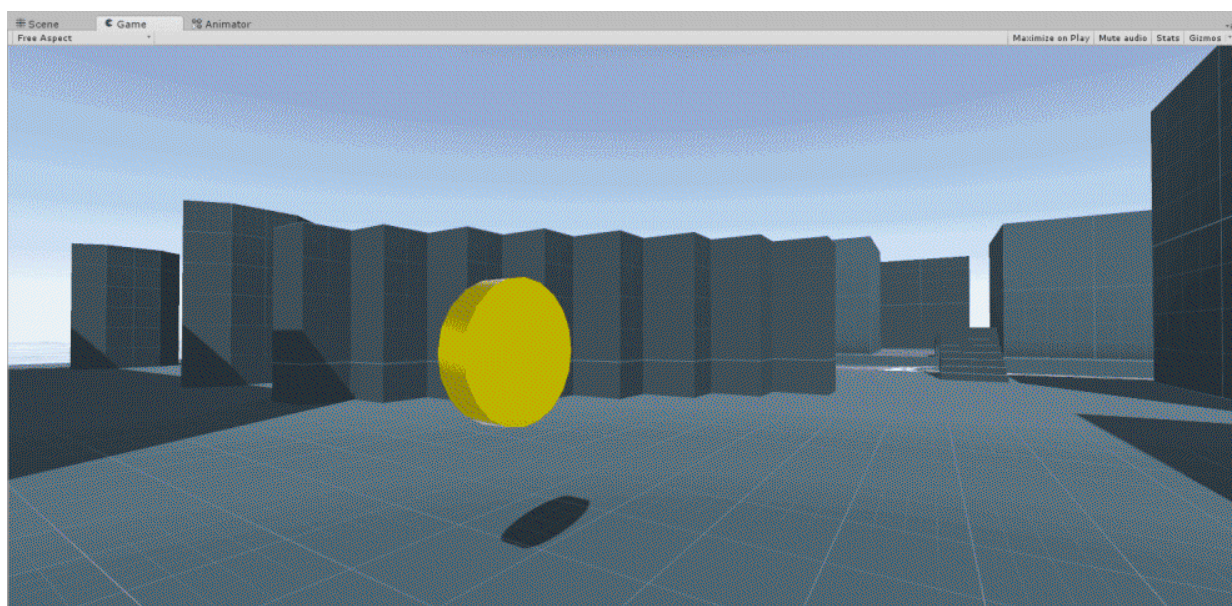


图1.1 金币采集游戏（最终完成效果）



本章和下一章介绍了这个金币采集游戏的开发过程，关于这个金币采集游戏的完整代码，可以在本书的配套文件Chapter01/CollectionGame目录下找到。

## 1.2 从头开始——Unity中的项目

当在Unity中创建一个新游戏时，如现在的金币采集游戏，都需要先创建一个新的项目。一般来说，Unity中的“项目”这个词就等同于“游戏”。在Unity中有两种方法来创建一个新的项目，但其实这两者并没有太大的差别。在打开的Unity的图形化编辑界面中，可以看到一个场景或者关卡，从应用菜单上依次选择“File | New Project”，如图1.2所示。这时Unity会询问是否对当前已经打开的项目进行保存，这时需要根据自己的实际情况来选择“Yes”或者“No”。在选择完“New Project”以后，Unity将会出现一个项目创建向导，根据这个向导的要求，就可以一步一步完成项目的创建工作了。

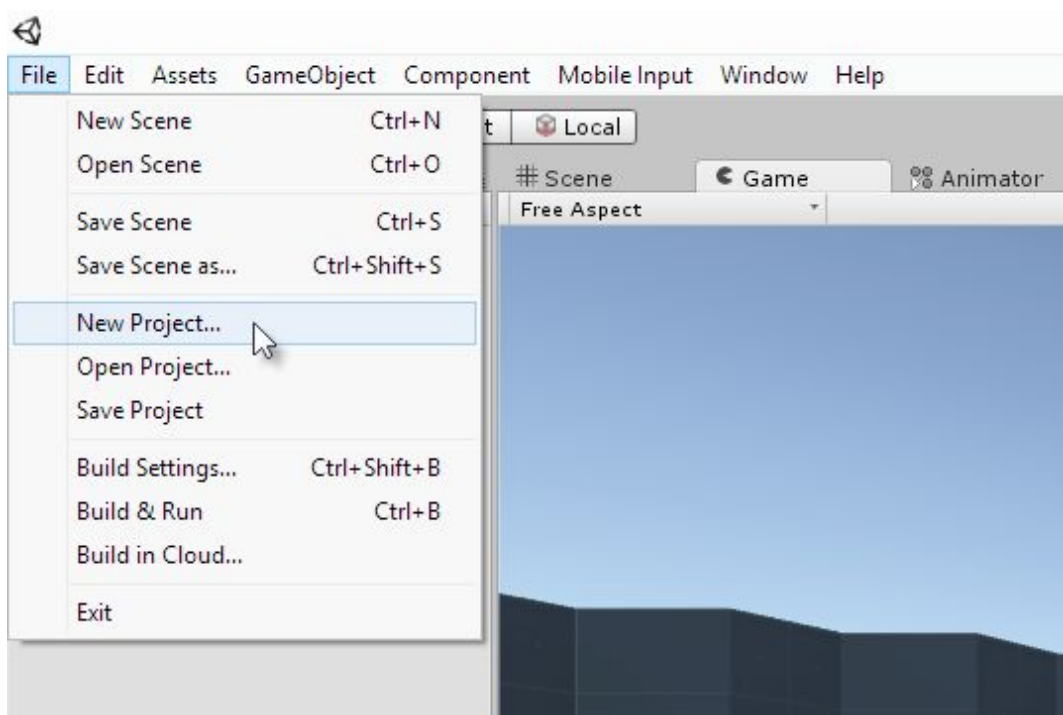


图1.2 使用主菜单来创建一个新的项目

如果是第一次启动Unity，则会看到一个欢迎的对话框，如图1.3所示。在这个对话框中，可以通过选择“NEW PROJECT”按钮来创建新的项目。

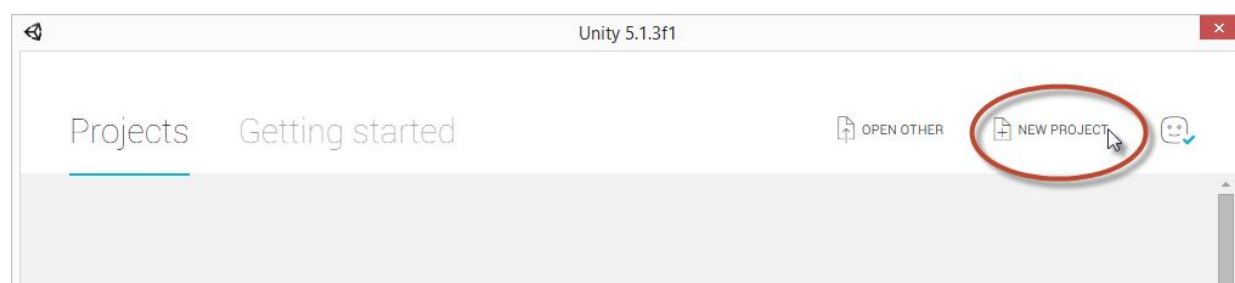


图1.3 Unity的欢迎界面

选择“NEW PROJECT”创建向导之后，Unity就会按照指定的一些基本设置来创建一个新的项目。此处需要填写项目的名称（例如CollectionGame），然后在电脑上选择一个用来保存自动生成项目文件

的文件夹。最后，注意3D和2D的这两个按钮，当单击3D按钮时，这意味着要创建的是一个三维游戏；如果单击的是2D按钮，这意味着将要创建一个二维的游戏，所有这些步骤都完成之后，需要单击“Create project”按钮来完成整个项目的生成过程，如图1.4所示。

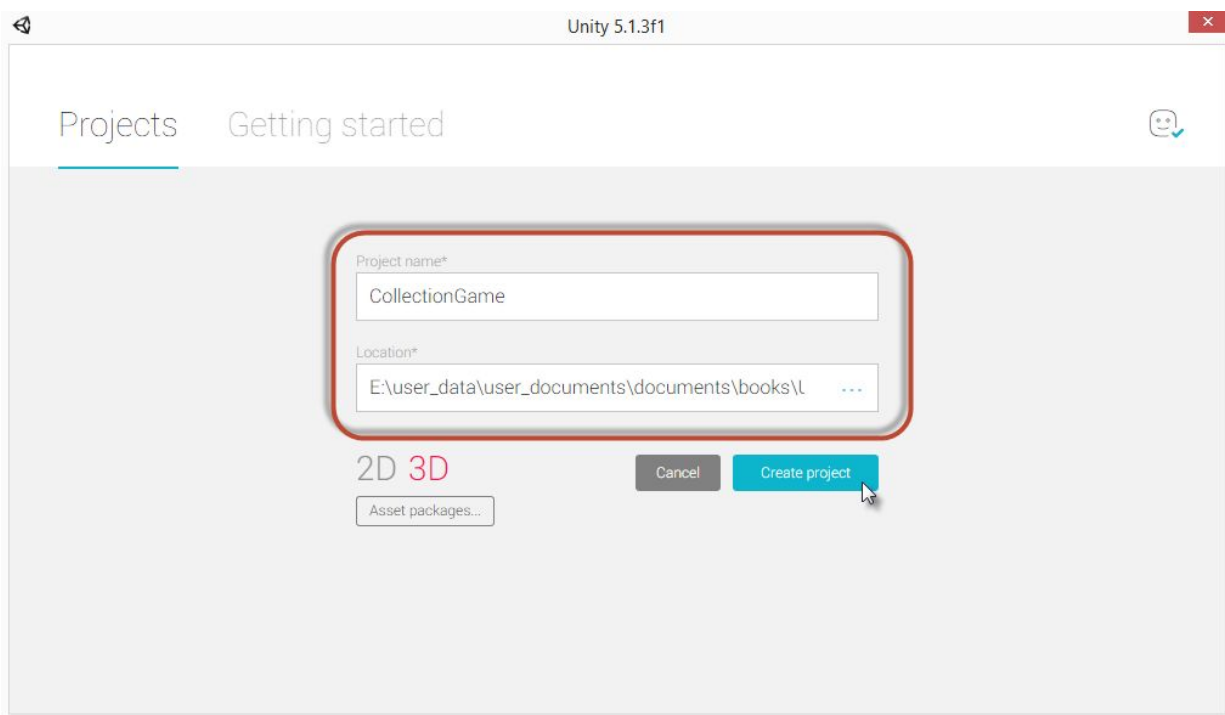


图1.4 创建一个新的项目

## 1.3 项目和项目文件夹

现在Unity已经创建了一个空白的新项目。这里就是开发一个新游戏的起点，在这个新建的项目中并不包含任何东西，没有任何的网格，贴图或者其他资源。这一点只需要检查一下位于Unity编辑器界面下方的项目（Project）面板区域就可以确定。这个项目（Project）面板中会显示项目文件夹中的全部资源，同时它 also 对应着一个本地驱动器上的文件夹，而这个文件夹就是在之前项目向导中所创建的。这个



文件夹如图1.5所示，现在也是空的。在这个游戏的开发过程中，还会使用到这个项目（Project）面板中的更多功能。

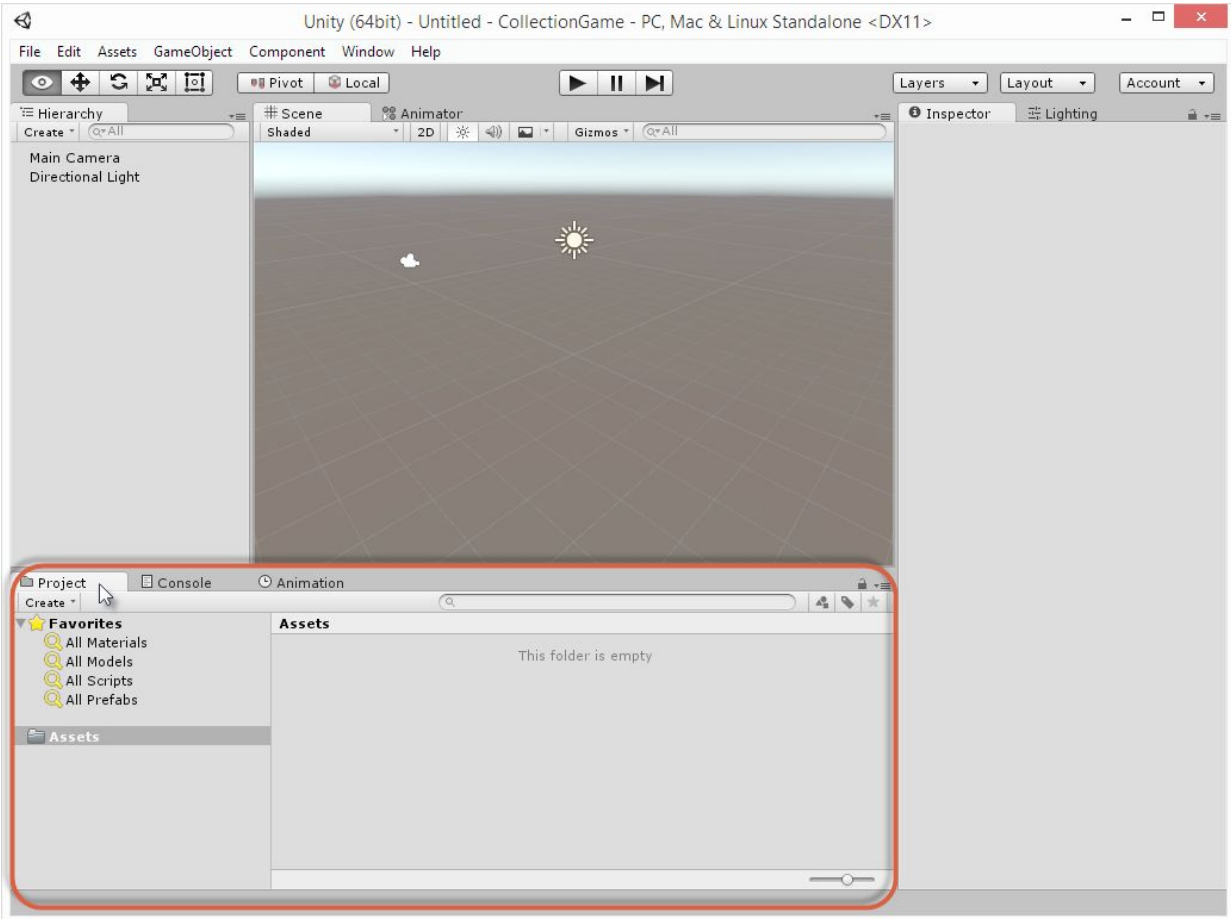


图1.5 位于Unity开发界面的底部的项目（Project）面板



如果现在使用的Unity开发界面的布局与图1.5所示的有些不同，那么可以选择将用户开发界面的布局进行重置，恢复为默认布局。只需要单击开发界面右上角的布局下拉菜单，然后再选择“Default”，见图1.6。

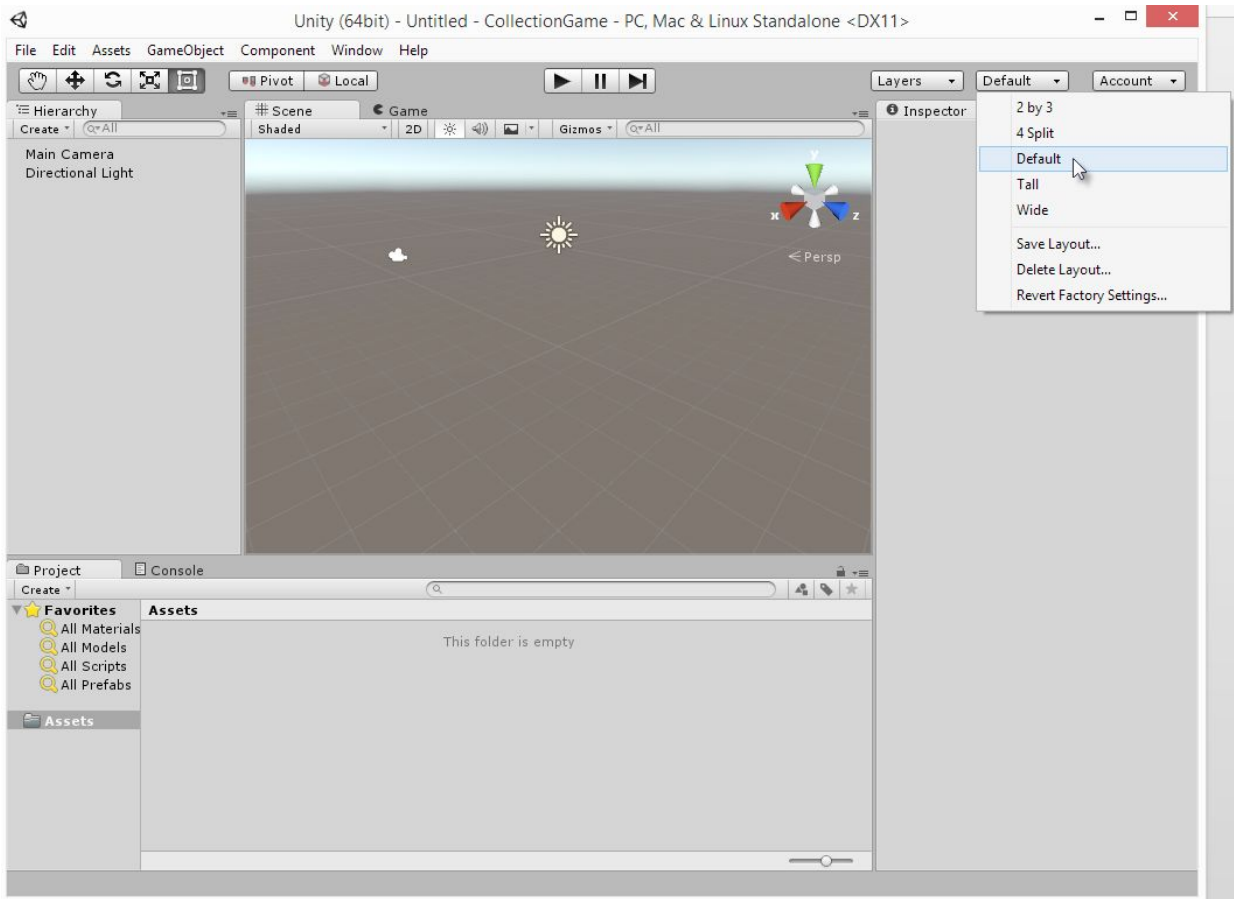


图1.6 切换成默认的开发界面布局

可以在Windows操作系统下的资源管理器或者Mac操作系统中的Finder直接查看到项目文件夹中的内容。要做到这一点，只需要在Unity编辑器上的项目（Project）面板中单击鼠标右键，这时就会显示出一个内容菜单，然后在这个菜单上选择选项“Show in Explorer”，这个过程如图1.7所示。



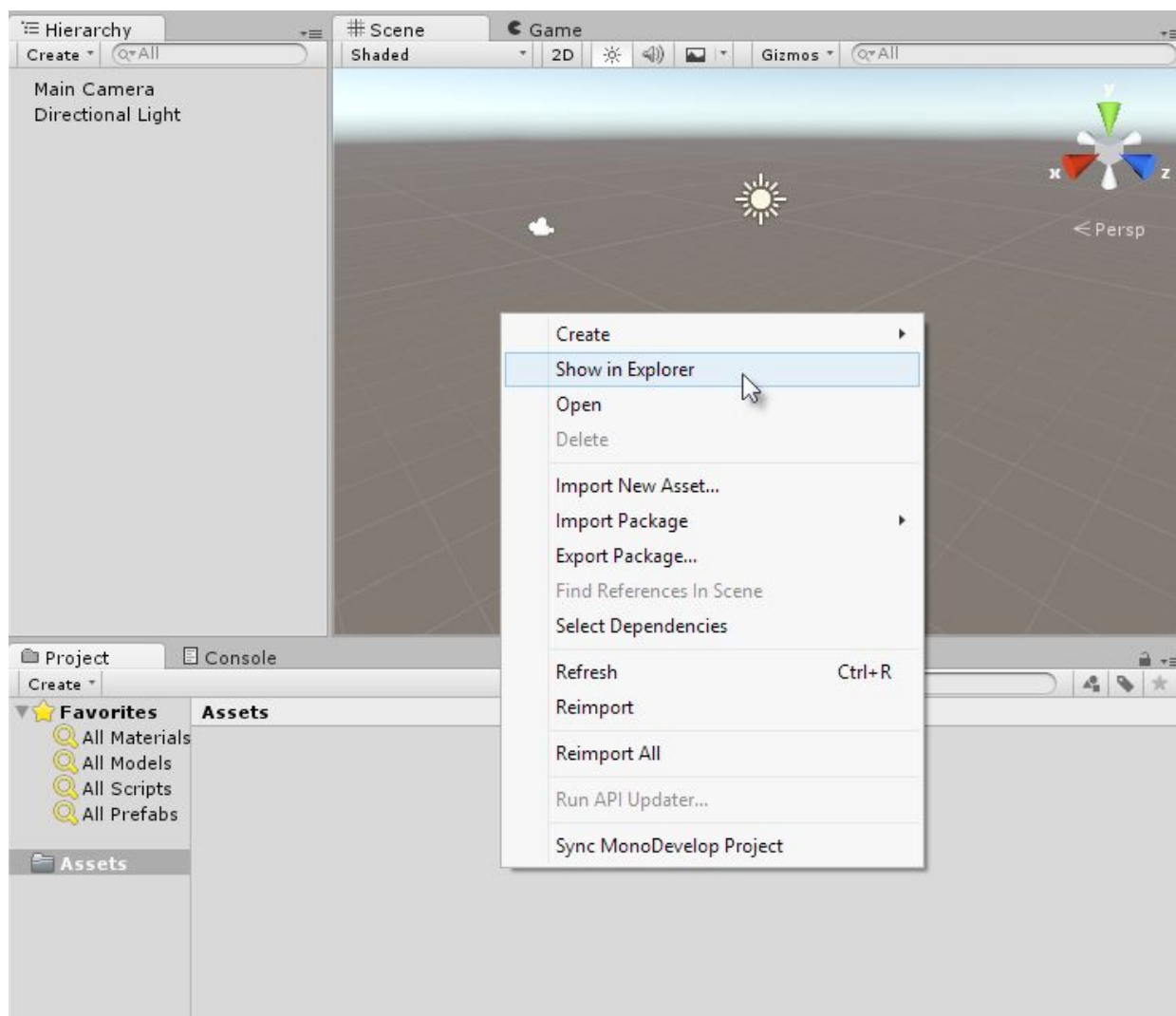


图1.7 使用项目（Project）面板显示项目文件夹

单击“Show in Explorer”可以在默认的系统文件浏览器中对文件夹中的内容进行浏览，如图1.8所示。以这种方式查看文件，可以更容易地实现对文件的检查、清点或者备份操作。但是，不要在Windows操作系统下的资源管理器或者Mac操作系统中的Finder中对文件夹中的内容进行修改，尤其是不要进行移动、重命名或者删除文件的操作，这样操作可能会导致Unity项目损坏。如果进行文件的删除和移动操作，就

在Unity编辑器中的项目（Project）面板中进行。只有这样操作，Unity才会同时对它的元数据进行更新，以确保项目能继续正常工作。

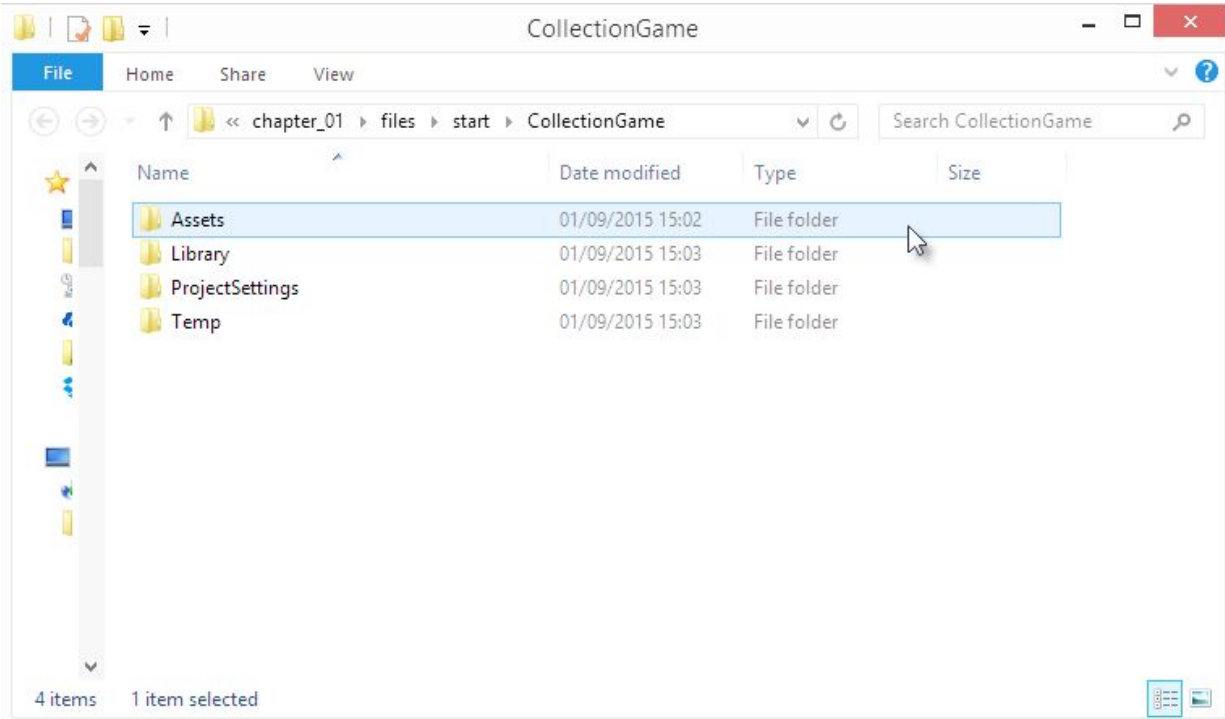


图1.8 从操作系统的文件浏览器中查看项目（Project）面板的内容



如果是从操作系统的文件浏览器中查看项目文件夹，就可以看到一些在项目（Project）面板中看不到的文件和文件夹，例如Library文件夹、Project Settings文件夹或者Temp文件夹。这些文件夹中保存的都是项目的元数据。这些文件并不是项目直接的组成部分，但是却包含一些额外的设置和首选项信息，这些信息保证了Unity能正常运行。注意，不要试图对这些文件夹和其中的文件进行修改或者编辑。

## 资源的导入

资源是游戏中的原材料，也就是组成游戏的模块。Unity中的资源包括网格（或者3D模型）、贴图、各种音乐和音效、场景，其中网格包括各种人物、道具、植物、建筑物等；贴图包括各种JPEG文件和PNG文件，这些图片决定了网格的外观；音乐和音效用来提高游戏真实性和气氛；场景（Scene）是一个网格、纹理、音乐、音效等协同工作而组成的独立系统。因此可以这样说，游戏不能没有资源，否则这些游戏看起来空虚而又无趣。基于这个原因，金币采集游戏也同样需要资源。毕竟这个游戏需要一个可以在其中进行移动和采集操作的环境。

Unity只是一个游戏开发引擎，而并非是一个资源开发软件。这意味着在游戏中需要的资源（例如各种人物、道具），通常是由一些设计者使用其他软件开发出来的。然后，这些设计者将这些制作好的资源导出，并传递给Unity。Unity则负责将这些资源有机地组合到一个游戏系统中。这些用来设计3D模型资源的第三方软件包括Blender（这是一款免费的软件）、Maya或者3DMAX。而对于纹理的创建，可以选择Photoshop或者GIMP（这也是一款免费的软件）。至于音效的制作，可以选择使用Audacity（这款软件同样是免费的）。当然除了这些软件以外，还有更多的选择。这些软件的具体内容并不在本书的范围内。通常，Unity引擎会认为已经有了可以导入到游戏中的资源。例如，在金币采集的游戏中将使用Unity所提供的资源。现在，将这些资源导入到这个项目中，为此，需要首先选择菜单栏上的“Assets”选项，然后在下拉菜单上选中“Import Package”，然后如图1.9所示依次选中“Characters”“ParticleSystems”“Environment”和“Prototyping”等选项。

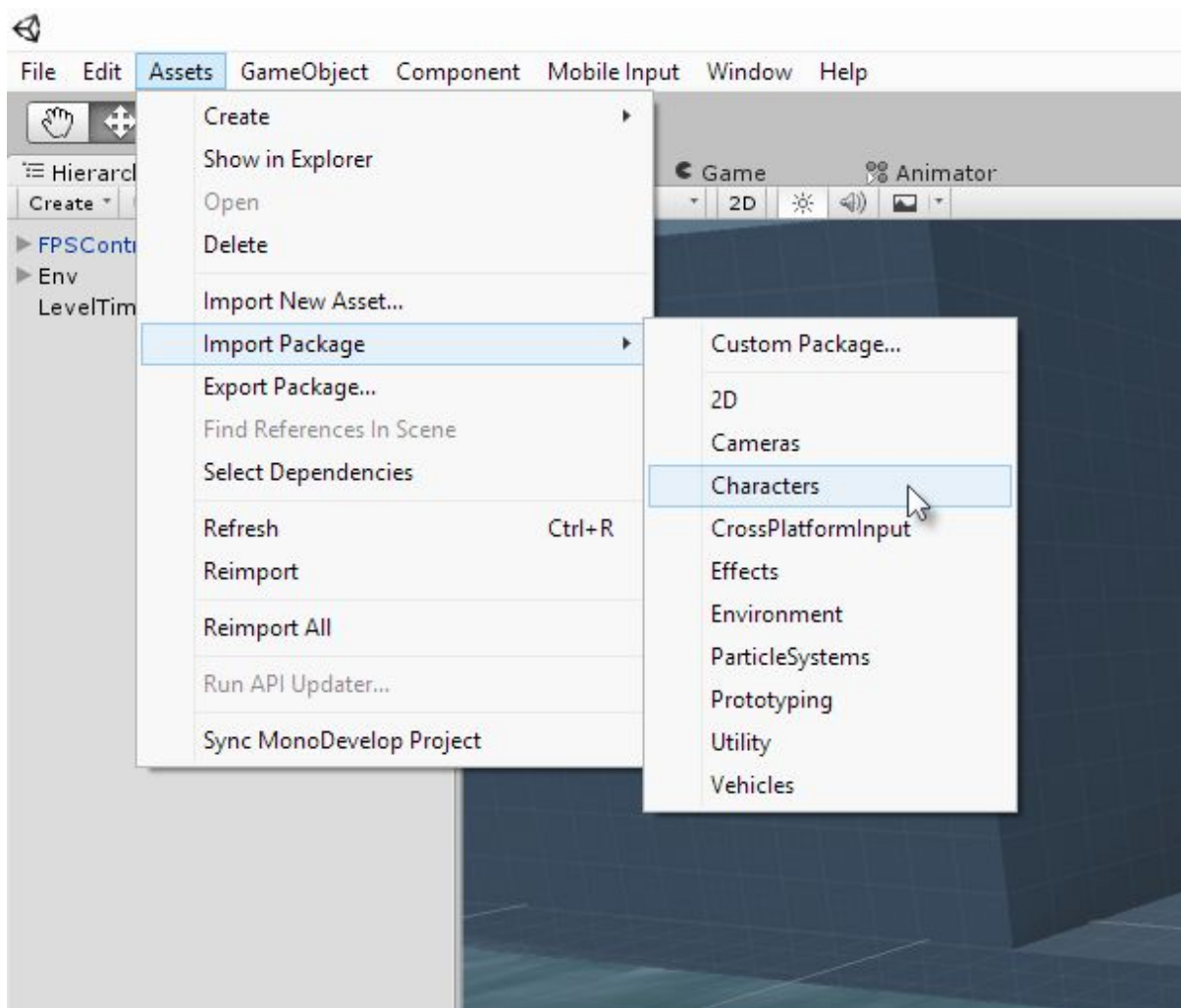


图1.9 使用资源包导入（Import Package）菜单完成资源导入操作

当在菜单中进行一个资源包的导入操作时，都会出现一个导入操作的对话框。在这个对话框中无需进行任何改动，保留里面设置的默认值，然后单击“Import”按钮即可。这个过程如图1.10所示。

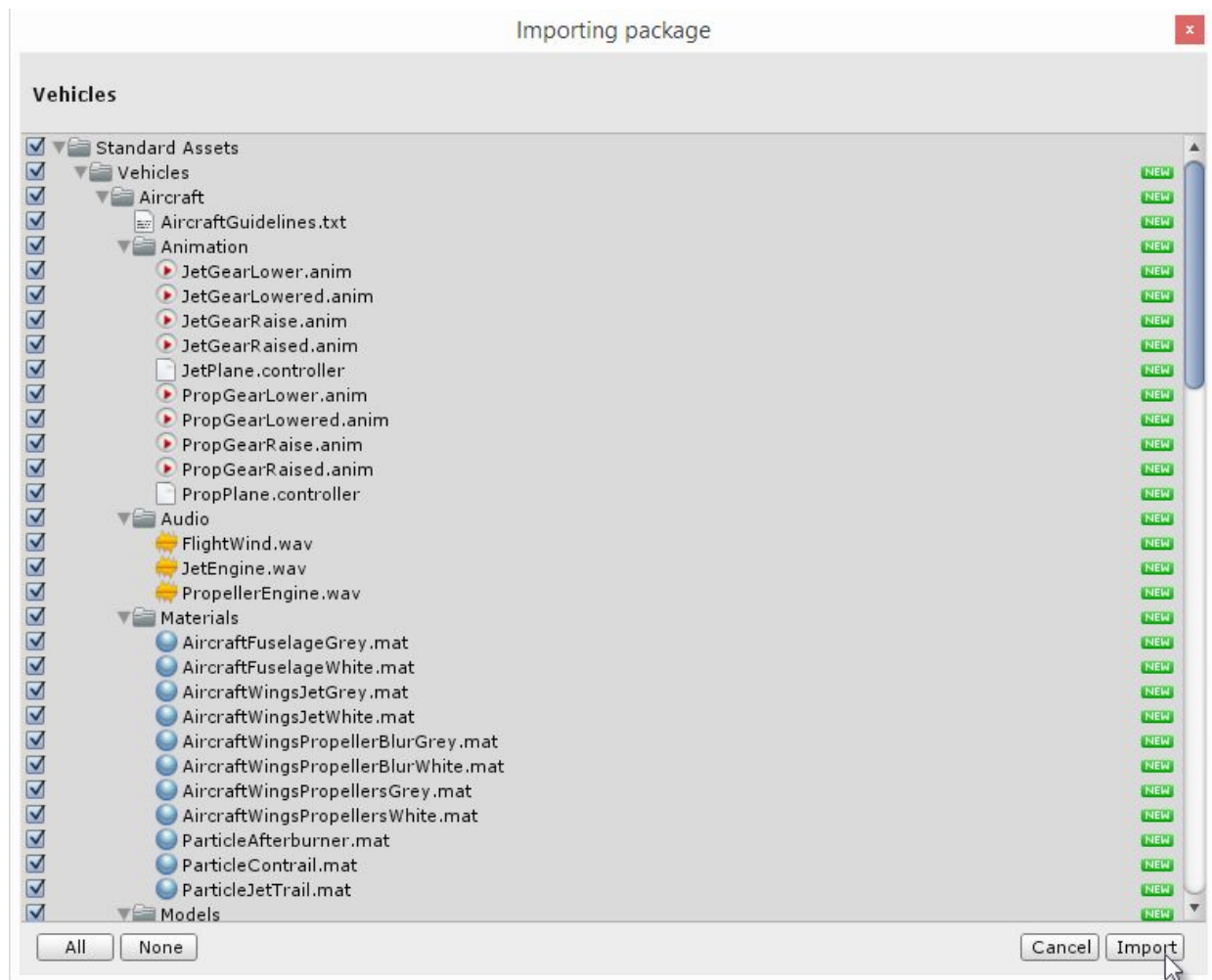


图1.10 选择需要导入的资源

默认情况下，Unity会从资源包（一个资源的库文件）中的所有文件解压缩到当前项目中。在导入操作结束后，各种各样的大量资源和数据将会添加到项目中，以后可以随时使用这些文件，它们都是原来文件的副本。所以，无论对项目中的哪个文件进行更改，都不会影响到原来文件。这些文件包括各种模型、声音、贴图等。图1.11所示为在Unity项目（Project）面板的编辑器中列出的这些文件。

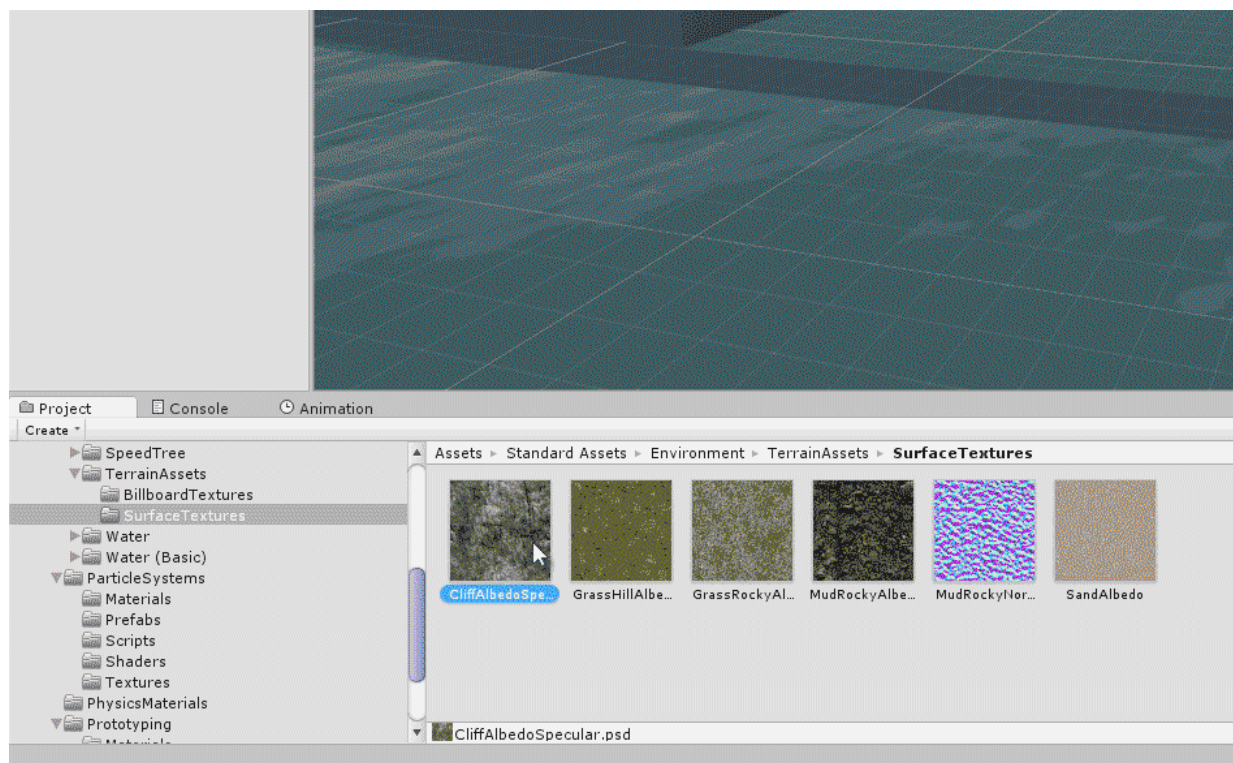


图1.11 从项目（Project）面板中对导入的资源进行浏览



当在应用菜单上依次选择了“Assets | Import”之后，如果什么资源包都没有，那么可以从Unity的主页[https:// Unity3d.com/](https://unity3d.com/)上下载和安装这些资源包。打开这个主页之后，首先选择“Additional Downloads”，然后在如图1.12所示的页面上选择“Standard Assets”。



# DOWNLOAD UNITY

Hello! We know you want to quickly download and start using Unity, so let's go!



[Release notes](#) [System requirements](#) [Unity 5 upgrade guide](#)

RELEASE DATE	VERSION	FILE SIZE	PLATFORM
24 AUG 2015	5.1.3	636KB	WINDOWS ▼
ADDITIONAL DOWNLOADS FOR WINDOWS ▼			
<a href="#">Unity Editor (64-bit)</a>			
<a href="#">Unity Editor (32-bit)</a>			
<a href="#">Built in shaders</a>			
<a href="#">Standard Assets</a>			
<a href="#">Example Project</a>			
<a href="#">Tizen support for editor</a>			

图1.12 标准资源包（Standard Assets）的下载

然而导入的资源并没有存在于游戏之中，没有出现在屏幕上，也不能做任何事情！它们只是被简单地添加到了项目（**Project**）面板，其实这些资源现在更像是一个库或者说是资源的仓库，可以从其中进行选择来完成游戏的建立。到目前为止，这些资源已经被内置到了Unity之中，在之后的章节中将继续使用它们，来完善金币采集游戏的功能。如果想了解关于每个资源的详细信息，可以使用鼠标单击选中这个资源，这时关于这个资源的详细信息就会出现在Unity编辑器右侧的检查（**Inspector**）面板中。检查（**Inspector**）面板是一个位于开发界面

右侧的属性编辑器，它是上下文敏感的，并会自动切换为显示所选对象的属性，如图1.13所示。

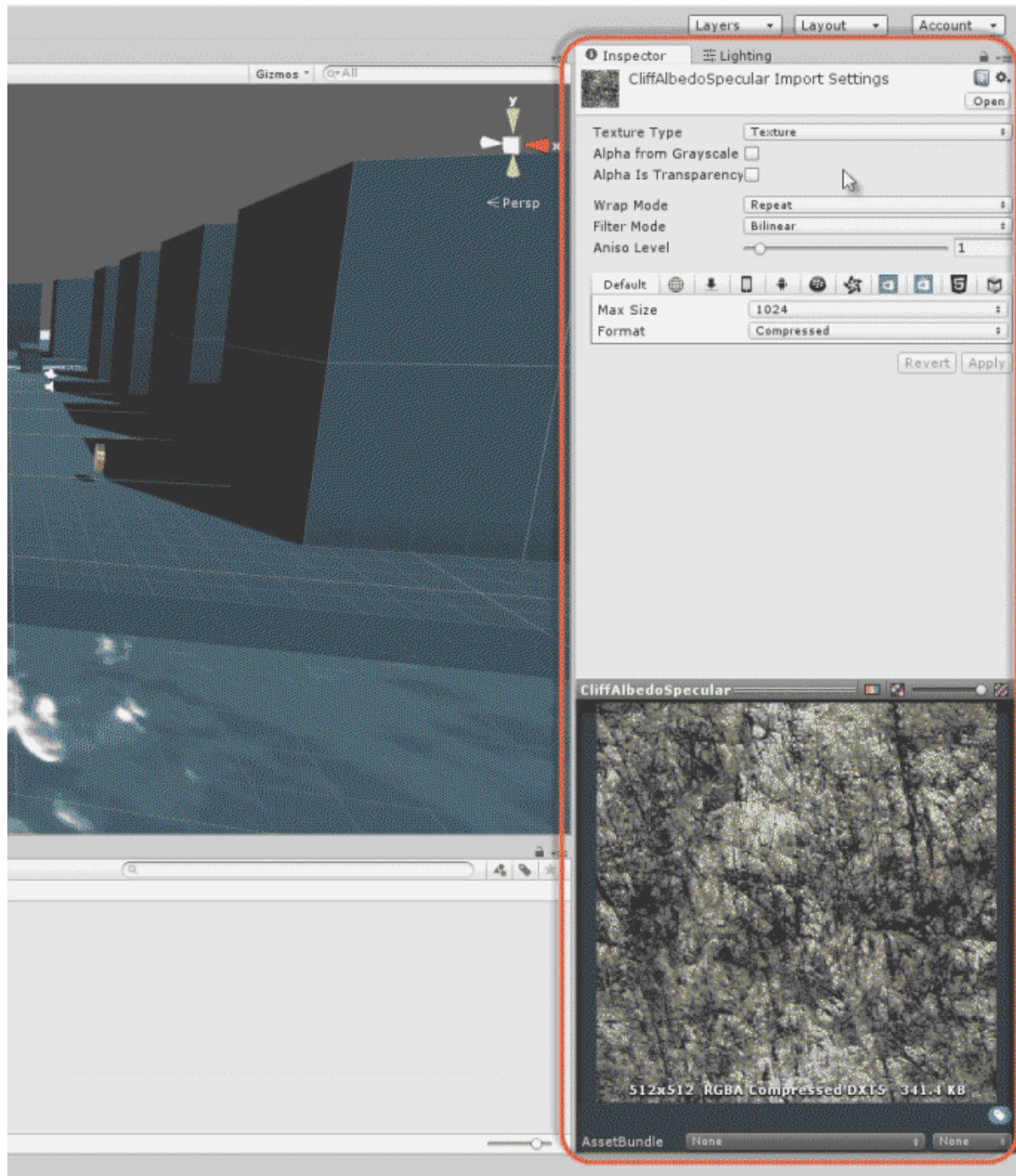


图1.13 检查（Inspector）面板显示了当前选中物体的所有属性



## 1.4 开始一个关卡

现在已经创建了一个Unity的项目，并且利用Unity的标准资源包导入了一个很大的资源库，包括一些建筑类网格，例如墙、地板、天花板、楼梯等，将使用这些资源建立第一个关卡（level）。记住，在Unity中，一个场景（Scene）往往也就意味着一个关卡。场景和关卡这两个词汇在这里是没有区别的，都是指一个三维空间，也就是游戏发生的时空。现在来创建一个金币采集游戏的场景，首先从应用程序菜单依次选择“File | New”，或者也可以在键盘上按下“Ctrl + N”组合键。当完成了这个操作之后，一个新的空白的场景就被创建好了。可以在占据了Unity开发界面最大部分的场景（Scene）选项卡中看到场景的预览，如图1.14所示。

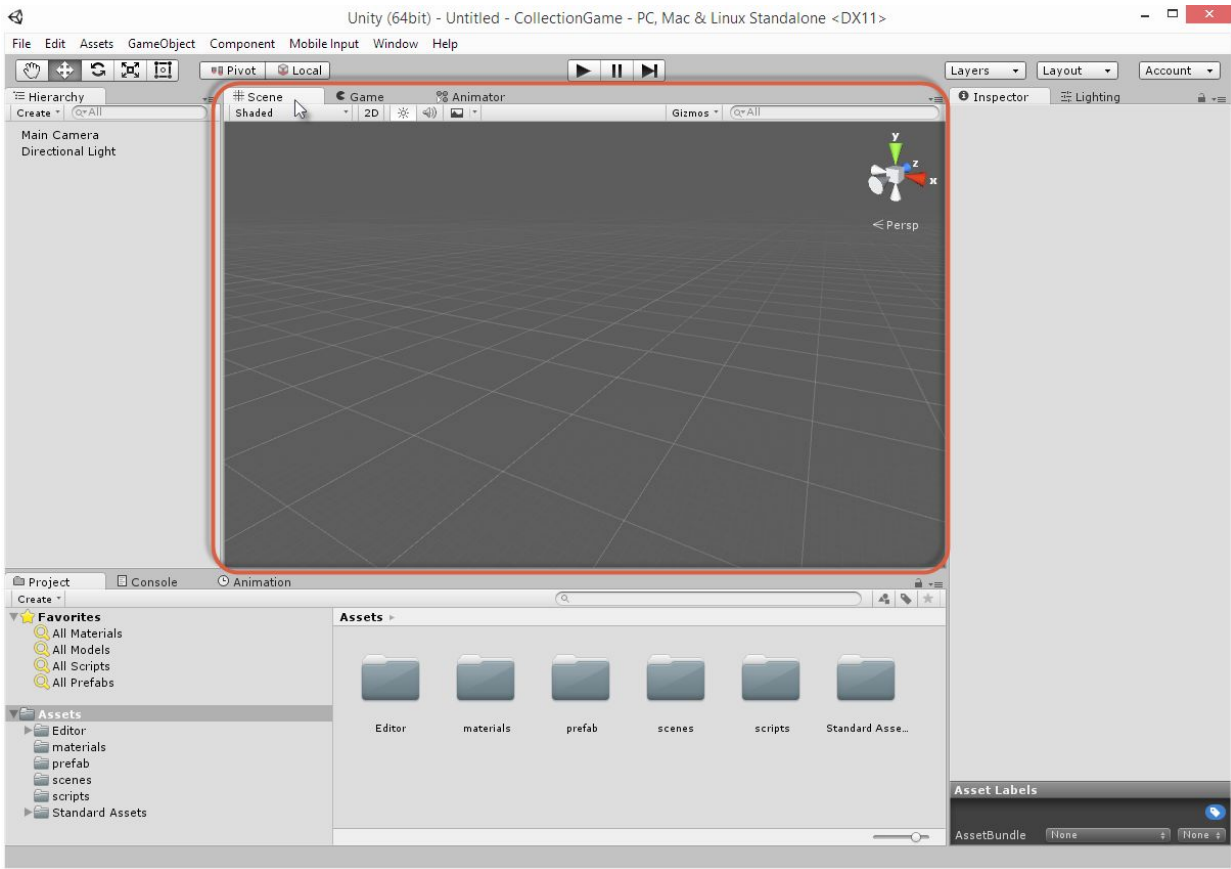


图1.14 场景（Scene）选项卡中给出了一个3D世界的预览



如图1.14所示，在Unity中的场景（Scene）选项卡以外的其他选项卡也是可见的，并且可以使用的。这些选项卡包括游戏（Game）选项卡和动画（Animator）选项卡。有时，选项卡的数目可能会更多。不过现在可以忽略除了场景以外的其他所有选项卡。场景（Scene）选项卡是为了能够在游戏的开发中简单快速地查看一个关卡。

每一个新的场景都是空的，严格来说几乎是空的。默认情况下，每一个新的场景都是由两个对象开始的，首先添加一个用来照亮其他

任意对象的光源对象（**Light**），然后是一个用来从特定角度对场景中的内容进行显示和渲染用的摄像机（**Camera**）。可以使用开发界面（见图1.15）左侧的层次（**Hierarchy**）面板来查看所有在场景中对象的完整列表。这个面板上显示了场景中所有对象的名称。在Unity中，游戏对象的含义就是在场景中存在的单一、独立的东西，无论它们是可见的还是不可见的，例如网格、光源、摄像机、道具等。因此，只有从层次（**Hierarchy**）面板才能看到游戏场景中的一切。

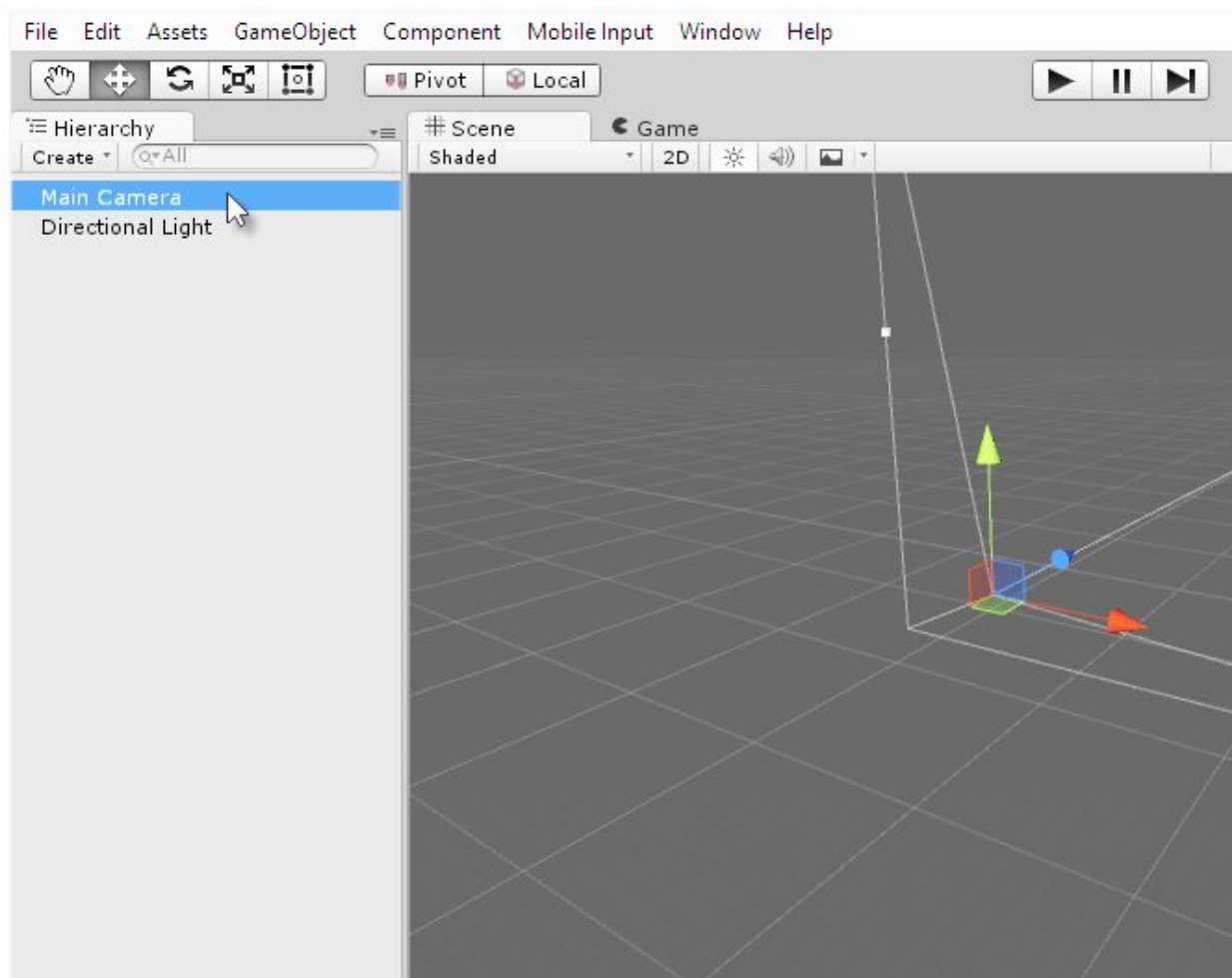


图1.15 层次（Hierarchy）面板



也可以通过在层次（Hierarchy）面板中单击游戏中对象的名称来选中它们。

接下来为这个场景添加一个地面（Floor）。要知道，游戏中的玩家总得站在一些东西上面。可以使用第三方的建模软件（如Maya、3DS Max或者Blender）来创建一个地面网格。不过，在之前导入过的标准资源包中就含有可以使用的地面网格，这是相当方便的。这些网格都是原型软件包（Prototyping Package）的一部分。如果想通过项目（Project）面板完成对这些资源的访问，可以双击打开“Standard Assets”文件夹，然后访问“Prototyping | Prefabs”文件夹。如图1.16所示，可以看到这些对象，并从检查（Inspector）面板中对它们进行预览。

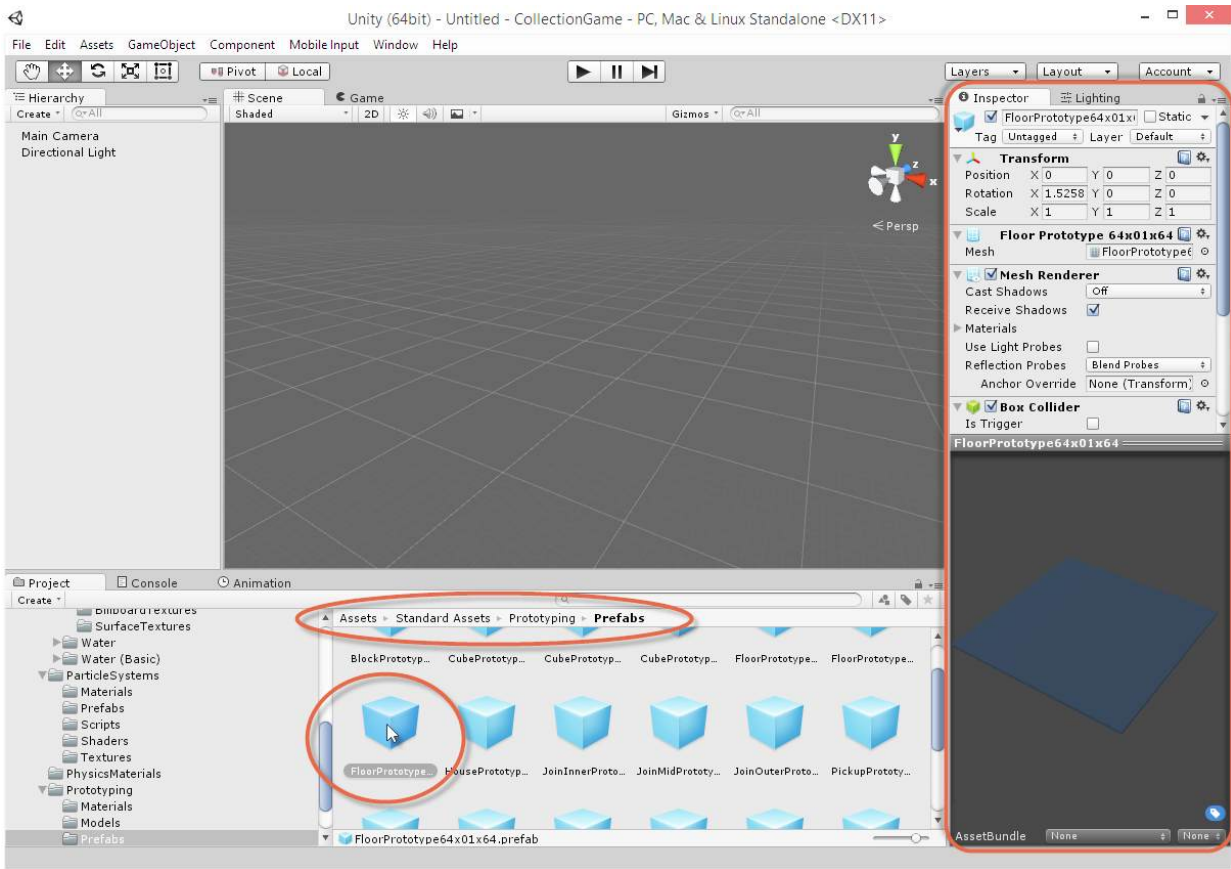


图1.16 包含了大量用于快速建立场景的网格的标准资源包/原型包



你也可以在应用程序菜单上依次选择GameObject|3D Object|Plane，这样可以更加方便地向场景（Scene）中添加地面（Floor）。不过，加进来的这个地面看起来一片灰暗，给人一种枯燥无味的感觉。当然，这个地面的外观是可改变的。正如我们在后面即将看到的一样，Unity中允许对物体外观进行改变。不过，在本书中，我们将从项目面板（Project panel）中的标准资源包里找到一个专门的地面网格（floor mesh）来使用。

图1.16所示的名为“FloorPrototype64x01x64”的网格十分适合作为一个地面，现在只需要简单地将项目（Project）面板（Panel）中的对象

拖曳到场景（Scene）视图中，然后释放鼠标即可完成这个网格的添加工作，如图1.17所示。当在进行这个操作时要注意，在三维空间中新添加了网格之后，场景（Scene）视图中发生的变化，同时也要注意到这个网格的名称也出现在了层次（Hierarchy）面板的列表中。

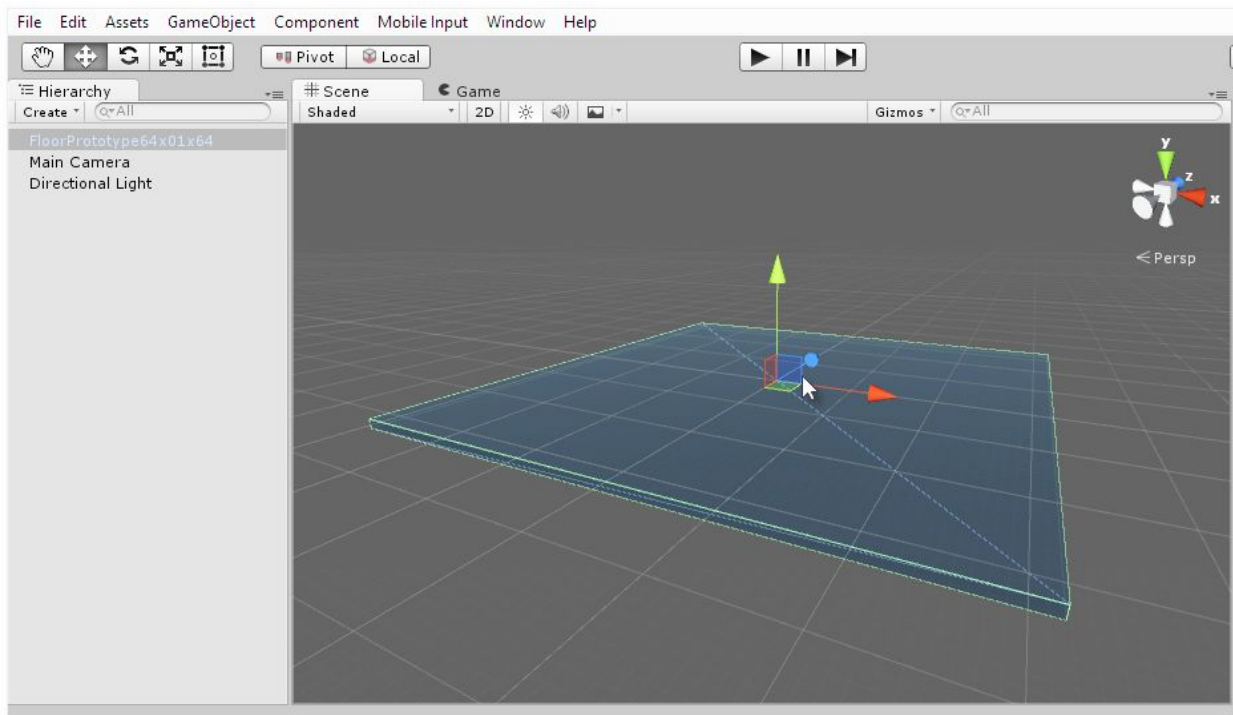


图1.17 将网格资源从项目（Project）面板中拖动到场景视图中

至此，现在项目（Project）面板上的网格资源已经被实例化为一个场景中的物体。这意味着一个项目（Project）面板上的副本已经作为一个独立的游戏对象加入到了场景之中。地面的实例（或者对象）依附于项目（Project）面板中的“floor”资源。但是反过来，这些资源却并不依附这些实例。这意味着当在场景中删除这些实例（或者游戏对象）时，这些资源并不会被删除。反之，当删除了这些资源时，场景中的这些实例就会被删除。如果场景中需要更多的地面，就可以多次从项目（Project）面板中向场景（Scene）视图拖动“floor”资源。每进行一



次这种操作，都会场景中添加一个单独的游戏物体。虽然这些游戏物体都依附于同一个“floor”资源，但是它们相互之间都是独立的（见图1.18）。

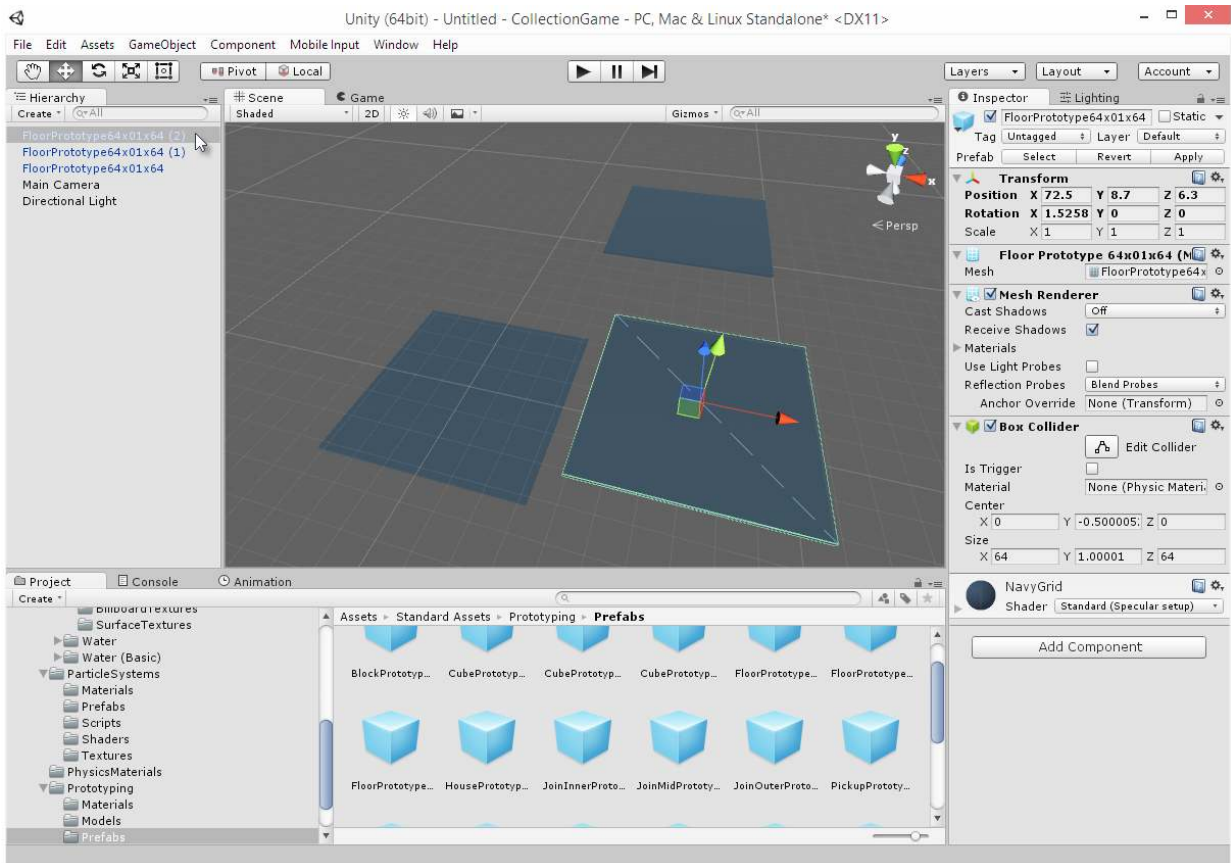


图1.18 向场景中添加多个“floor”网格

实际上，并不需要这么多重复的地面，所以先把它们删除。删除这些地面的方法是先在场景（Scene）视图中逐个单击这些“floor”对象，然后在键盘上按下删除键。另外记住，也可以通过单击层次（Hierarchy）面板上这些游戏物体的名字，然后按下删除键。不管使用的是哪一种方法，如今在场景中都只剩下了唯一一个“floor”对象。现在还有一个问题，那就是这个“floor”对象的名字问题。在层次（Hierarchy）面板上可以清楚地看到这个“floor”对象的名字

是“FloorPrototype64x01x64”，这个名字很长，意义又不够明确，而且调用的时候又不方便。因此应该将这个名字改为一个更容易管理的，而且意义更明确的，这样才能使工作更加有条理，也更容易管理。Unity中有很多种可以将对象重命名的方法，首先选中对象，然后在其检查（Inspector）面板中的名称字段处输入新的名字。这里将这个对象重命名为“WorldFloor”，如图1.19所示。

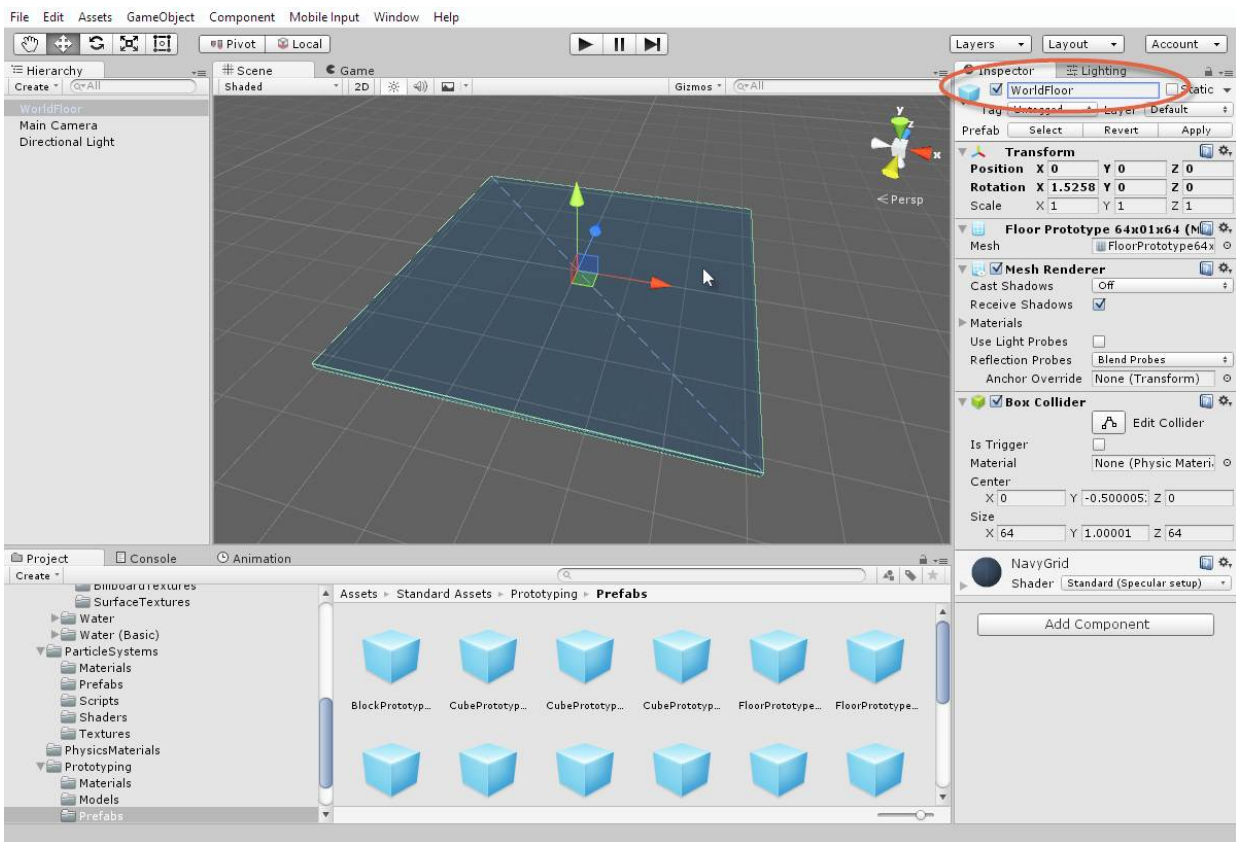


图1.19 对“floor”网格进行重命名

## 1.5 变换和导航

到现在为止，一个地面网格已经建立好了，但是仅仅这一个游戏对象还是很无趣的，还需要向其中添加更多的内容，例如建筑物、楼



梯、柱子或者更多的地面。否则，玩家在这个游戏中就没有能进行探索的世界了。在开始创建之前，首先要先确认现在的地面处在了整个世界的中心位置。场景中的任何一个位置都有一个唯一的坐标，这个坐标（ $x,y,z$ ）值指的是距离世界中心点（原点）的距离。在检查

（Inspector）面板中可以看到当前选中对象的坐标。事实上，在Unity中，一个对象的位置（Position）、角度（Rotation）和尺寸（Scale）同属于一个名为变换（Transform）的类别。位置表示的是一个对象距离3个坐标轴的距离。角度表示一个对象绕着它的中心轴旋转的角度。尺寸表示一个对象应该缩小到多小或者扩大到多大。默认的尺寸指的就是对象正常的大小，尺寸为2表示扩大到原来两倍的大小，尺寸为0.5表示缩小到原来的一半，以此类推。这样，一个对象的位置、角度、尺寸共同构成了它的变换属性。如果想改变一个选定对象的位置，可以在 $x$ 、 $y$ 和 $z$ 位置字段输入新的值。例如将一个对象移动到游戏世界的中心，就可以输入（0,0,0），如图1.20所示。

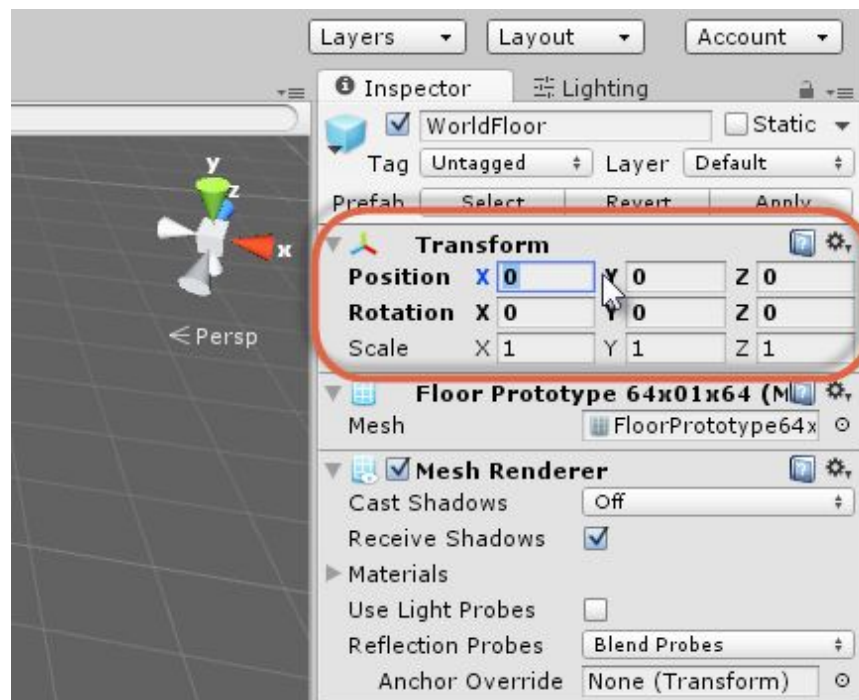


图1.20 将一个对象移动到游戏世界的中心点

如上所示，通过输入适当的值，当然这个值要在系统许可的范围内，就可以为对象指定准确的位置。不过，使用鼠标来直接移动对象往往更为直观。现在就来完成这个操作，首先添加第二个**floor**对象，这个**floor**对象的位置要远离第一个**floor**对象。先从项目（**Project**）面板上拖曳一个**floor**对象到场景中，然后单击这个新的**floor**场景以选中它，然后按下键盘上的“**W**”键，或者单击编辑器界面顶端的变换工具图标来切换到变换工具。变换工具允许对场景中的对象进行重新定位，如图1.21所示。

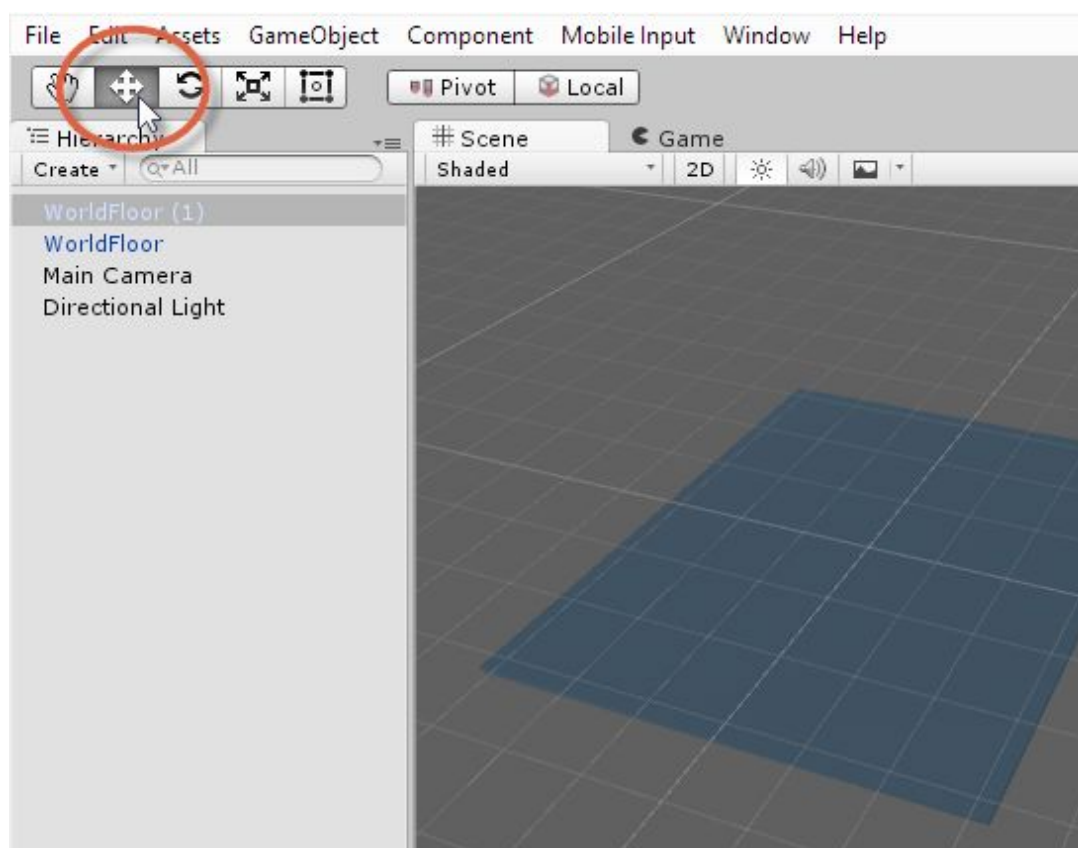


图1.21 打开变换工具

当选定一个游戏对象，并且打开变换工具以后，一个标识（Gizmo）就出现在游戏对象的中心部分，在场景（Scene）选项卡中可以看到标识（Gizmo）其实是由3个不同颜色的彩色轴组成的。其中，红颜色的轴代表x轴，绿颜色的轴代表y轴，蓝颜色的轴代表z轴。如果想移动一个游戏对象，首先要将光标悬停在这3个轴中的一个轴（或者两个轴之间的平面）上面，然后单击并按住鼠标，同时向目标方向拖动。可以反复地进行这个操作，将游戏对象移动到指定的位置。现在就用这个方法来拖动第二个floor对象，使其远离第一个floor对象，如图1.22所示。

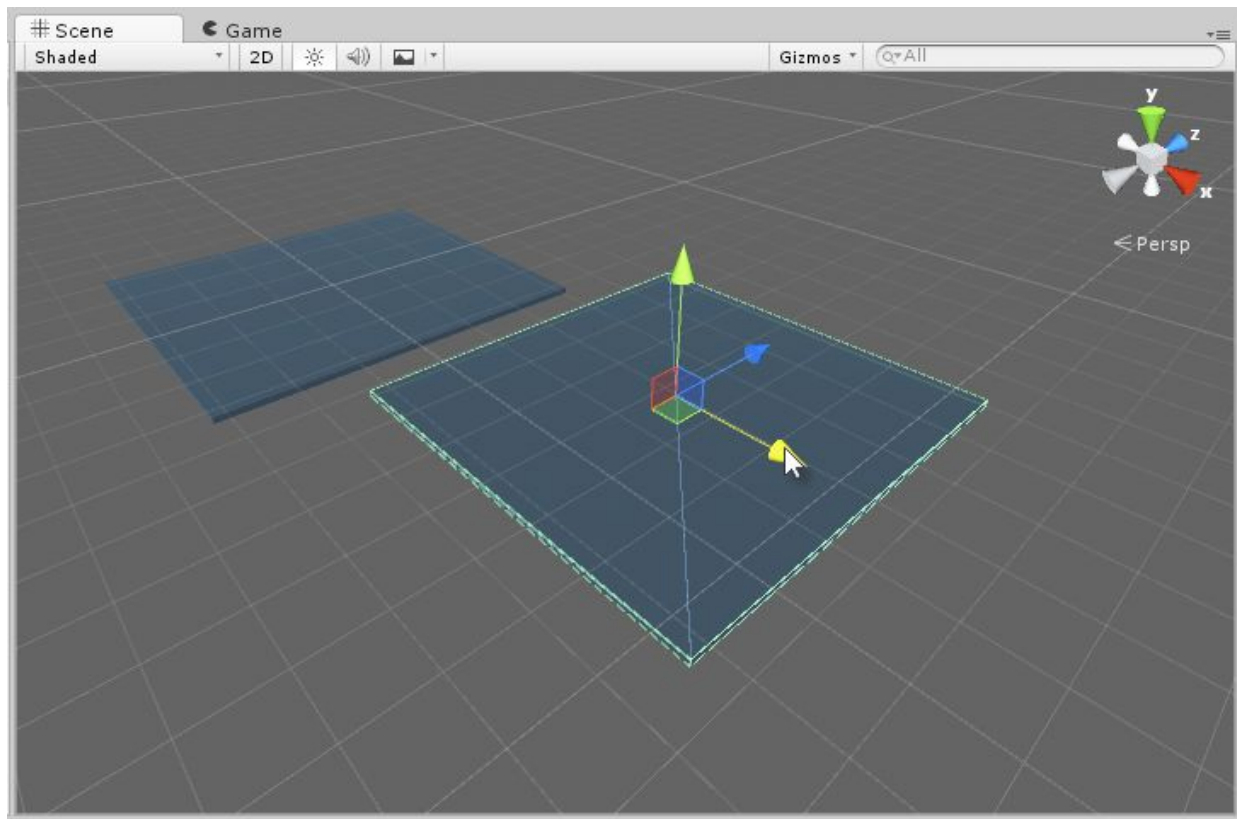


图1.22 使用标识（Gizmo）来完成对一个游戏对象的变换操作

也可以使用鼠标来完成对游戏对象的转动和缩放。在键盘上按下“E”键可以启动旋转工具，按下“R”键可以启动缩放工具，或者也可以在编辑器界面的上方单击它们对应的工具栏图标来启动这些工具。当工具启动之后，一个标识（Gizmo）就出现在这个对象的中心部分，可以通过单击和拖动鼠标来将这个对象旋转到指定角度，或者缩放到指定尺寸，如图1.23所示。

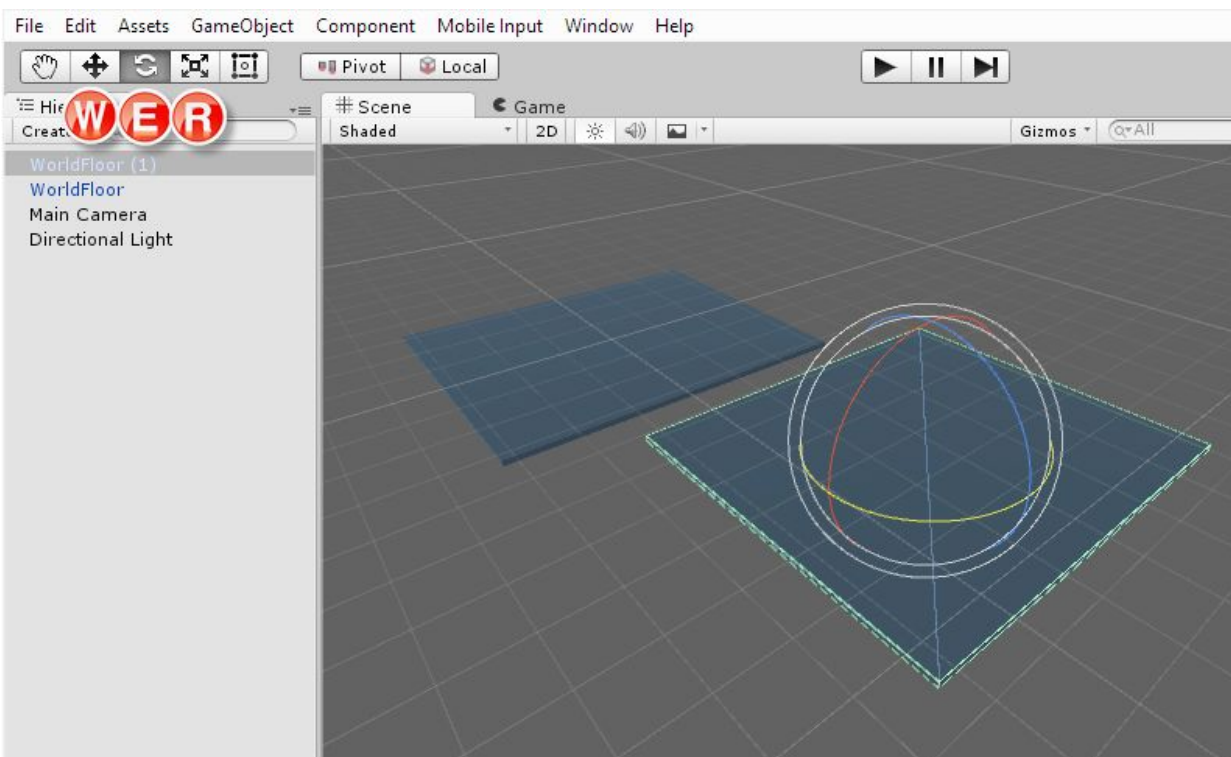


图1.23 旋转和缩放工具的选择

在Unity中进行开发时，使用键盘和鼠标对物体进行快速的移动、旋转和缩放是非常重要的。所以，最好习惯使用键盘的快捷键来完成这些操作，而不是总用鼠标去单击工具栏。不过，除了要对游戏对象进行移动、旋转和缩放以外，可能还需要频繁地变换自己的视角，这样才能从不同的位置、角度和视角对这个世界进行观察。这意味着必

须要经常改变场景中的摄像机。当要看清楚一个游戏对象时，就会需要进行放大或缩小的操作。如果想准确地将游戏对象对齐并结合在一起，就需要改变观察的视角。如果想完成这些操作，就需要充分地将鼠标和键盘结合在一起使用。

如果想靠近或者远离正在观察的对象，只需要向上或者向下滑动鼠标滚轮，就可以实现放大或者缩小目标对象，过程如图1.24所示。

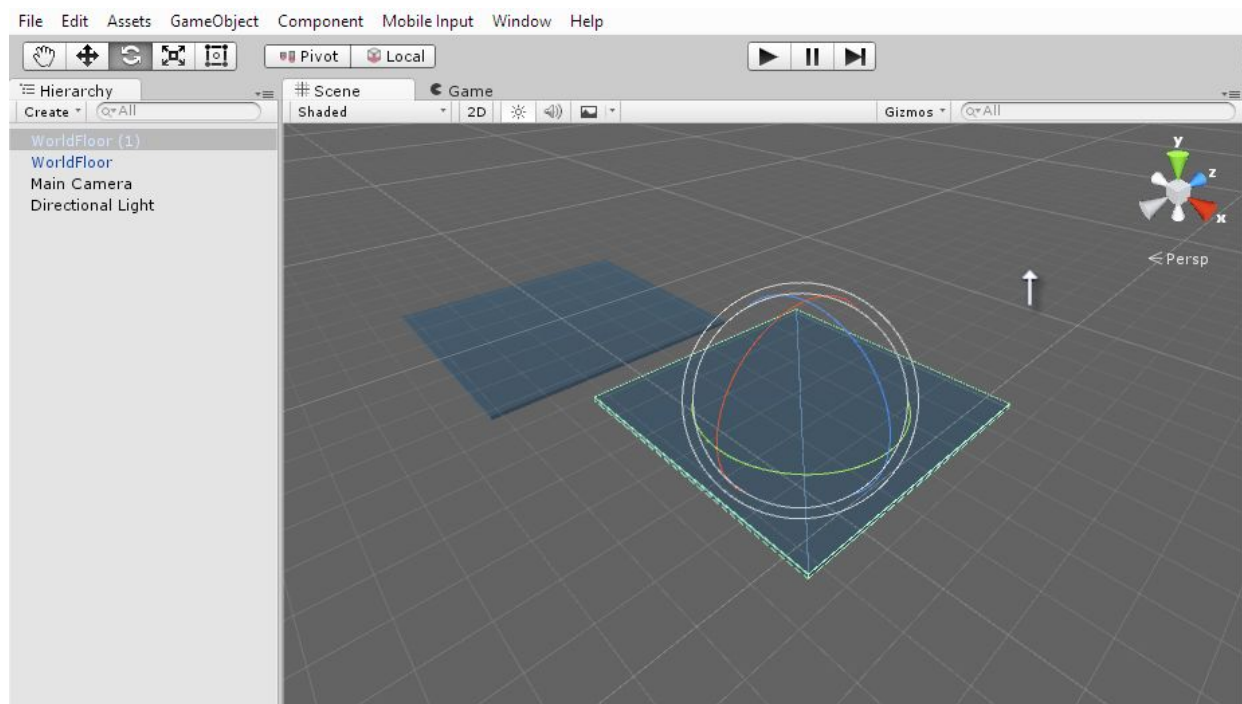


图1.24 放大或者缩小操作

按住鼠标的中键然后将鼠标向适当的方向移动，如向上移动，这样就可以将场景（Scene）视图向上移动。同样，向另外3个方向移动鼠标也可以将场景（Scene）视图向这3个方向移动。或者也可以在应用工具栏中（或者使用键盘上的Q快捷键）激活移动工具，如图1.25所示。当这个移动工具处于激活状态的时候，就可以十分简单地单击并



拖动整个场景（Scene）视图。注意，这个操作并不会改变场景的大小，而只是沿着向左、向右或者向上、向下的方向滑动摄像机。

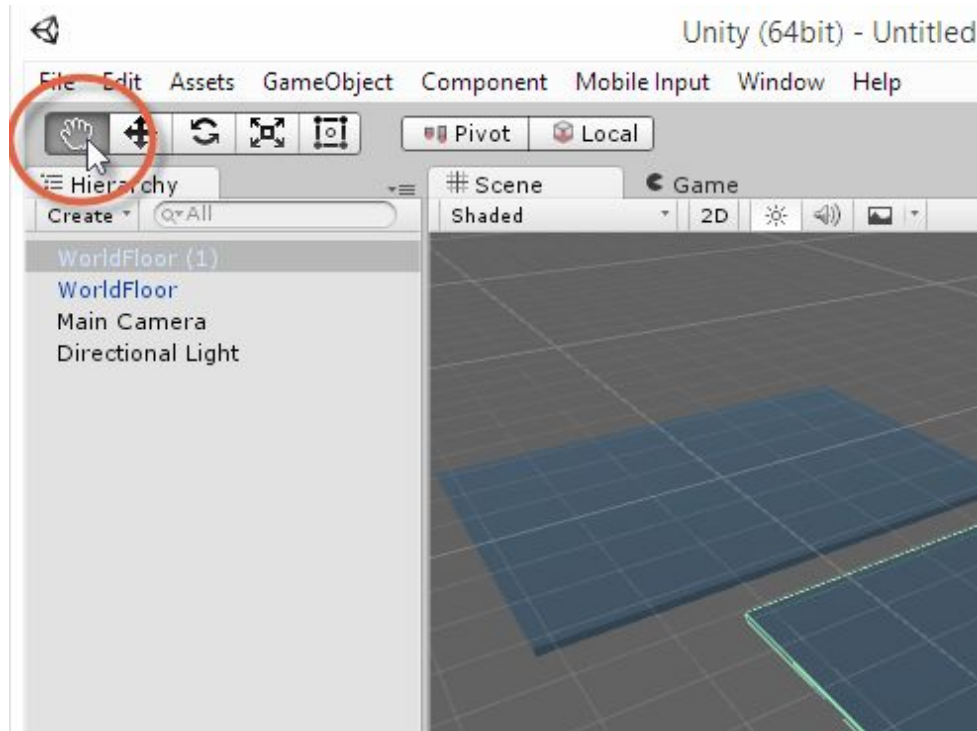


图1.25 激活移动工具

有时在构建关卡时，可能会找不到所需要的对象。例如，此时摄像机可能正在关注一个与目标完全不同的地方，这个地方并不是所要单击或者看到的。如果遇到了这种情况，必须要改变相机的视角来找到目标对象。如果将这个目标对象处在整个场景的中心，就必须不断地进行改变位置和旋转视图的操作。但是如果要自动地完成这一切，可以有一种更简单的方法，那就是从层次（Hierarchy）面板上选中这个对象的名字。然后，按下键盘上的“F”键，也可以直接在层次（Hierarchy）面板上直接双击来完成这个操作，如图1.26所示。

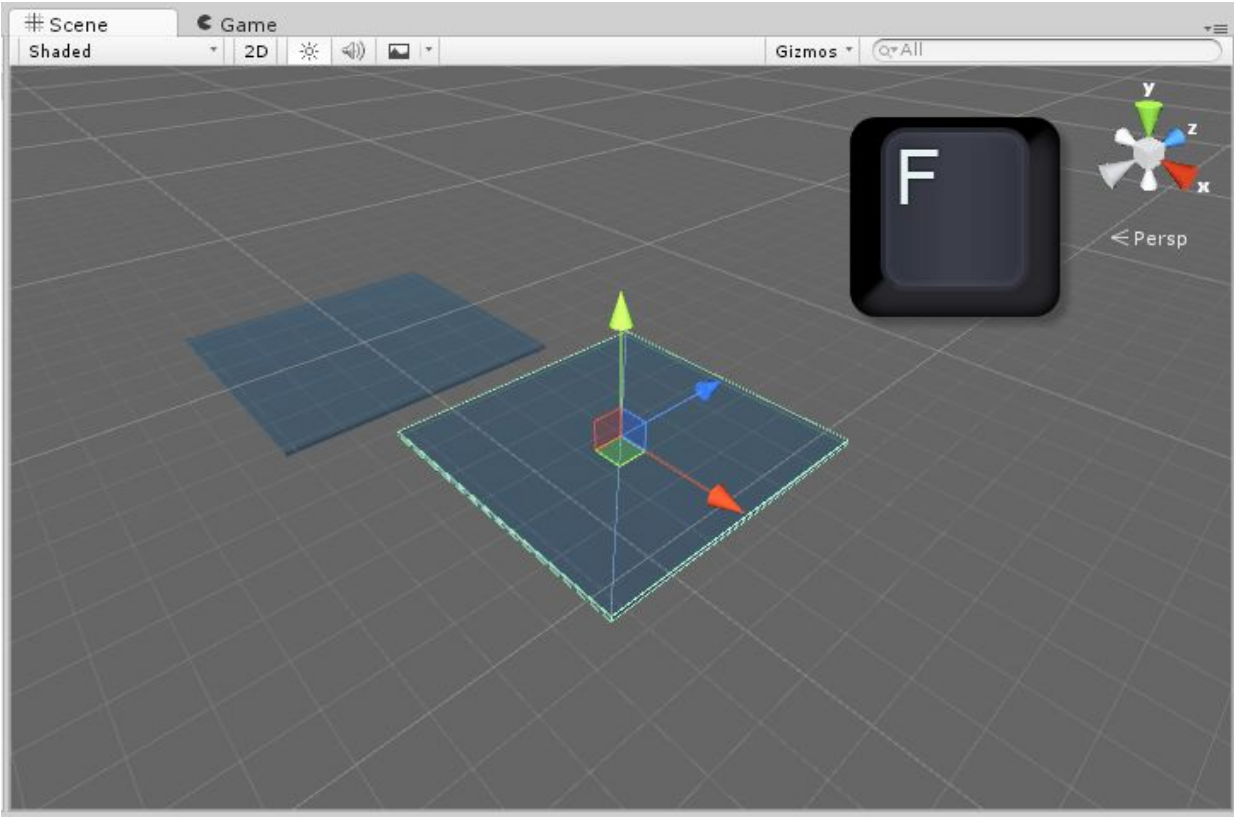


图1.26 一个选定的对象

当选中了一个游戏对象之后，可能会经常性地对这些对象进行旋转操作，以便能从所有的重要视角来对其进行查看，只需要单击鼠标，并在拖动的时候按住键盘上的“Alt”键，就可以完成视角的转动，如图1.27所示。

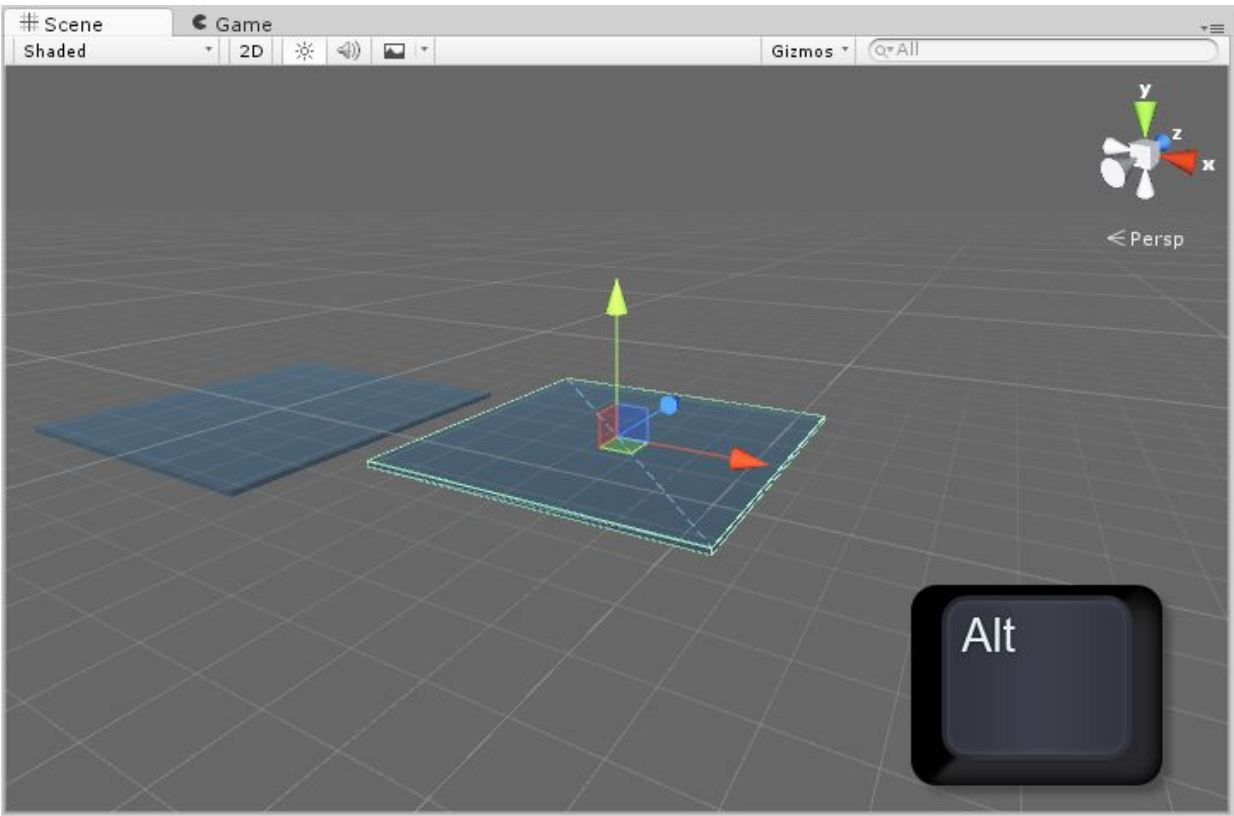


图1.27 绕着一个选中的游戏对象进行旋转

最后，在场景（**Scene**）视图中使用第一人称视角控制是十分实用的，像在玩一款第一人称视角的游戏一样，这将有助于你以一种身临其境的方式来体验整个现场。如果想实现这个功能，可以在按下鼠标右键的同时使用键盘上的“W”“A”“S”“D”来控制前进、后退或者左右转动。使用鼠标来控制头的方向。另外，还可以在运动的同时按下“Shift”键来提高运动的速度，如图1.28所示。



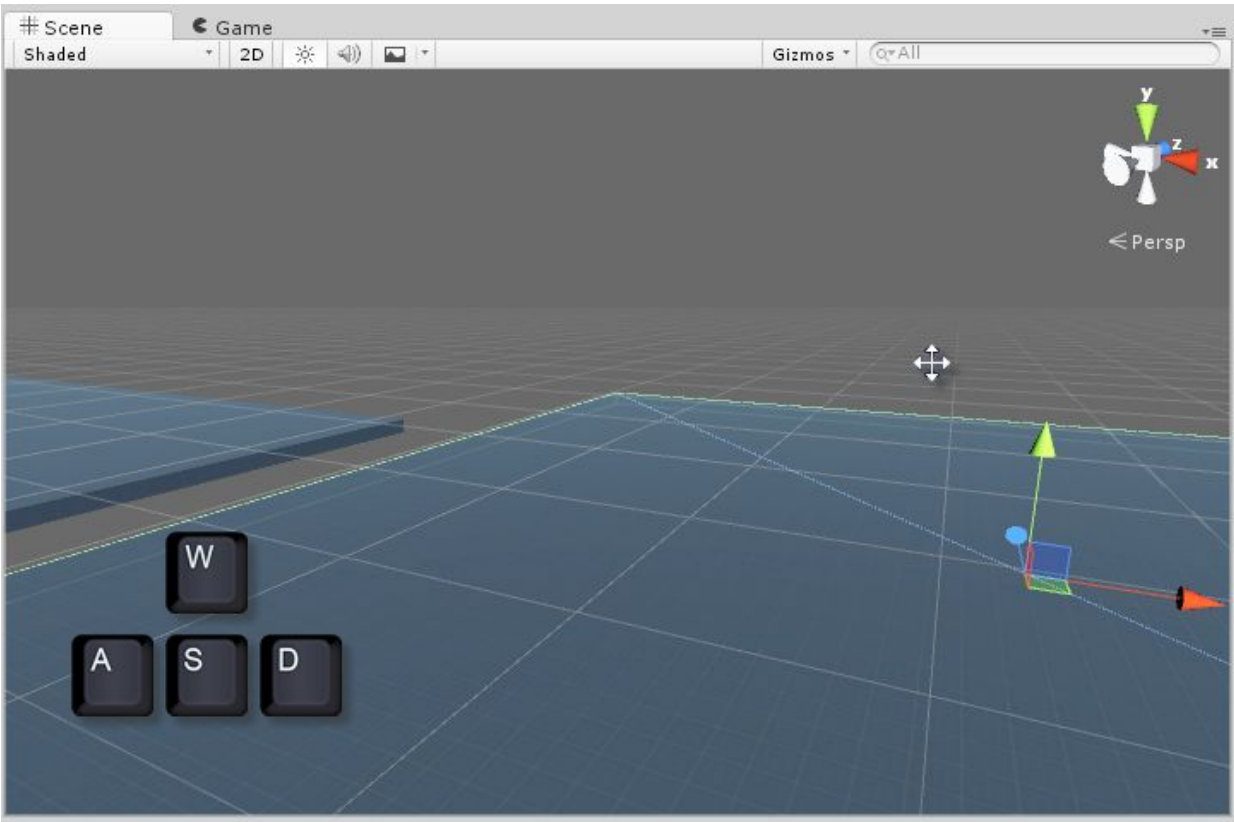


图1.28 使用第一人称视角控制

这里之所以选择对各种进行变换和导航控制的方法进行学习，就在于掌握了这些知识之后，你就可以以任何方式来定位和移动目标对象，也可以从任何的位置和角度来查看整个世界。如果想快速地建立一个高品质的游戏关卡，做到这一点是十分重要的。所有的这些以及一些其他的控制方式都将贯穿于这本书的场景创建和Unity开发之中。

## 1.6 场景的建立

现在已经成功地实现了对象的属性变换和在场景视图中的导航。接下来可以开始完成金币采集游戏的第一个关卡了。首先将空间中的两个floor网格对象分开，在它们之间留一个大的裂缝，将来会在这个

裂缝上创建一个桥梁。玩家将可以通过这个桥梁往返于两个如同岛屿一样的floor网格对象之间。可以使用变换（Translate）工具（W）来移动物体，如图1.29所示。

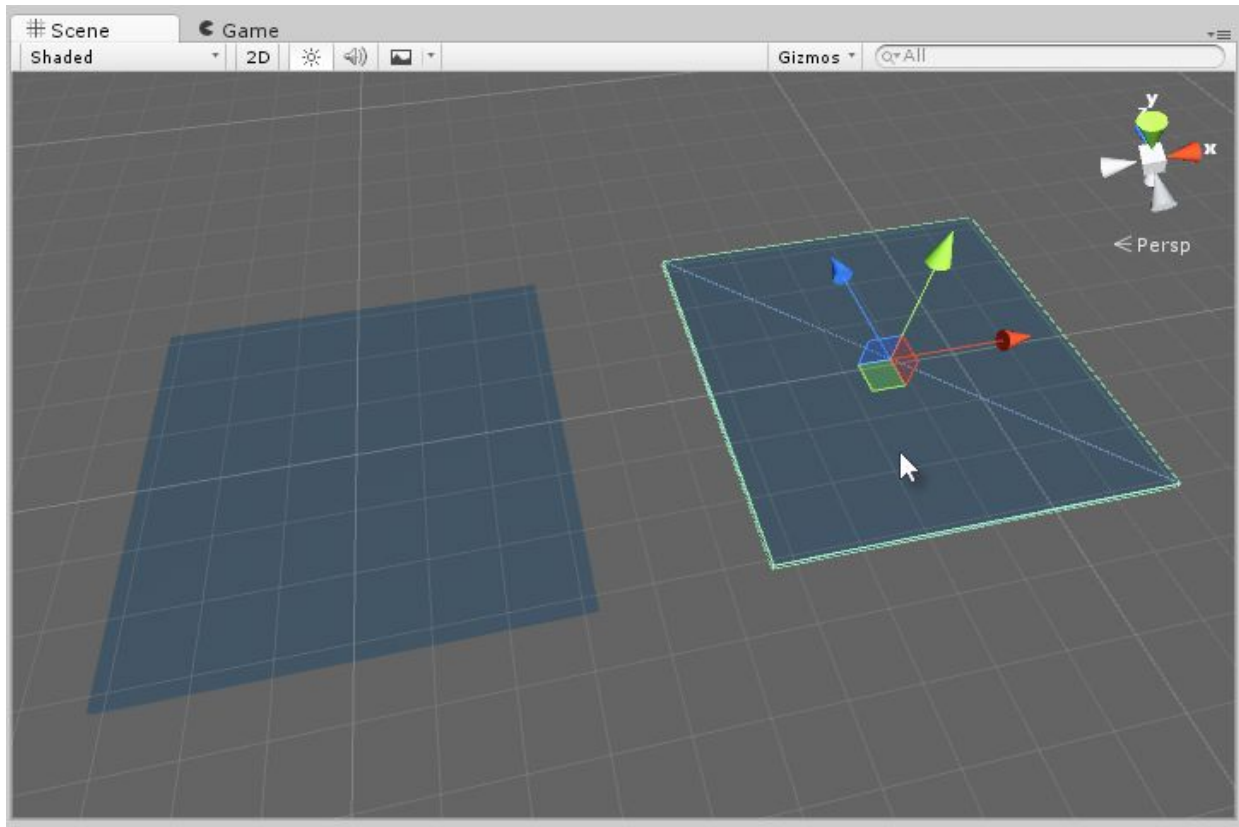


图1.29 将两个floor网格对象分隔成独立的岛屿



如果创建更多的floor对象，可以使用之前用过的方法，从项目（Project）面板处拖动一个网格资源到场景视图中。另外，也可以将在场景视图中选中的对象进行复制，复制的方法是在选中目标对象之后，同时按下键盘上的“Ctrl+D”组合键，这两种方法都可以达到目的。

接下来，要向场景中添加一些道具（Props）和障碍物（Obstacles），首先向floor网格对象上添加一些房屋（House）对象，可以按照“Assets | Standard Assets | Prototyping | Prefabs”这个目录找到房屋对象（HousePrototype16x16x24），如图1.30所示。

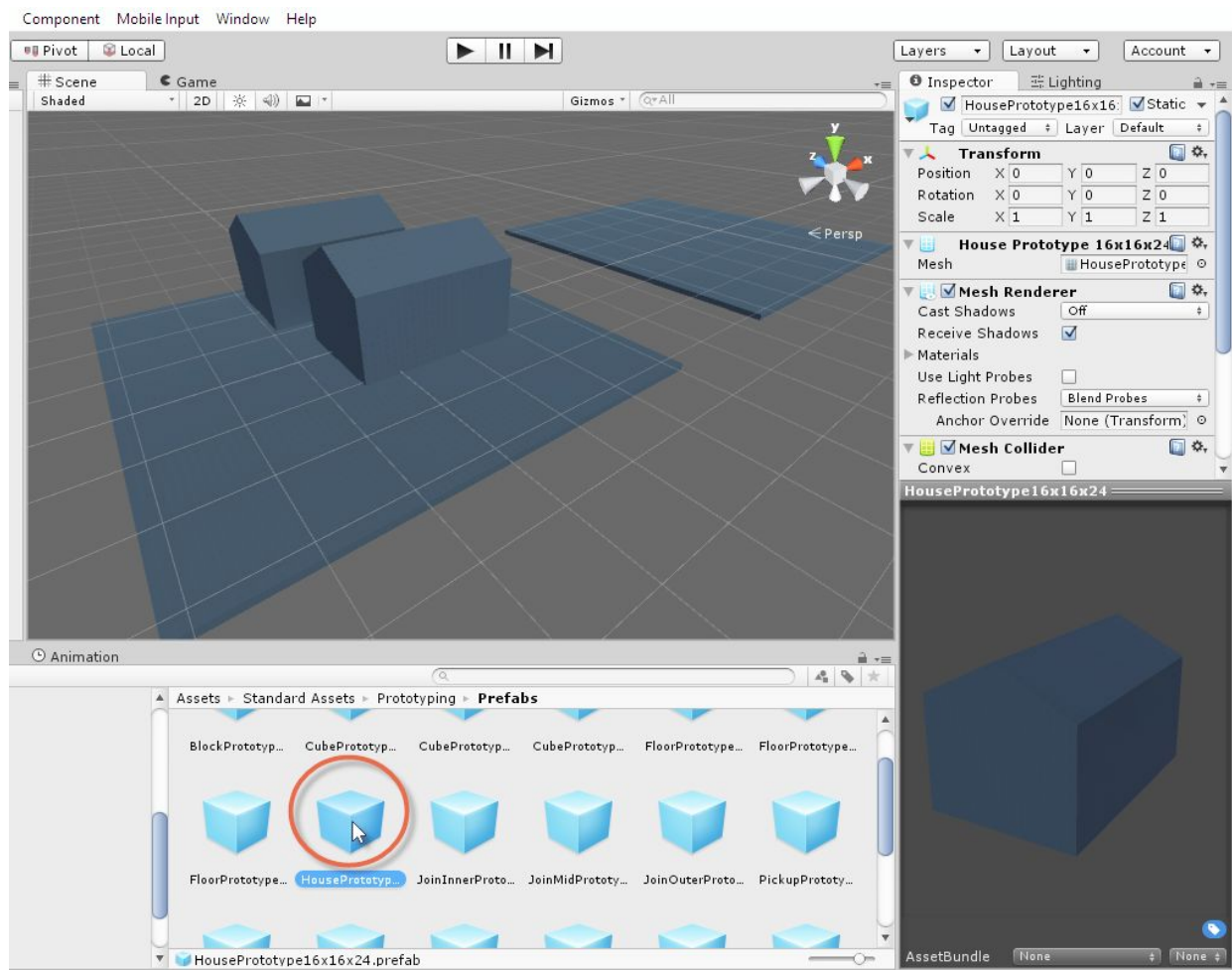


图1.30 向场景中添加房屋道具

当将一个房屋拖动到场景中之后，这个房屋可能会与地面的底部完美的对齐，但是也有可能不会，如果对齐了，那你的运气还真是不错。但是我们不可能每次都有这么好的运气，一个专业的游戏开发者应该依靠自身的技能，而不是运气。不过别担心，在Unity中，可以轻

松地使用顶点捕捉（Vertex Snapping）来将任意的两个网格对象进行对齐。这个功能的工作原理是使两个游戏中的对象通过将它们的顶点定位到同一个位置来实现对齐操作。

这里以图1.31为例，在这张图中房子歪歪扭扭地坐落在floor对象之上，我们很自然地希望将它和floor对象进行水平对齐或者角对齐。要实现这个效果，我们首先要选择房子（House）对象，这一点可以通过单击或者在层次面板中选择这个对象来完成，要注意进行选中操作的应该是要移动的那个对象，而不是那个用来对齐的参照物（此处特指floor对象）。

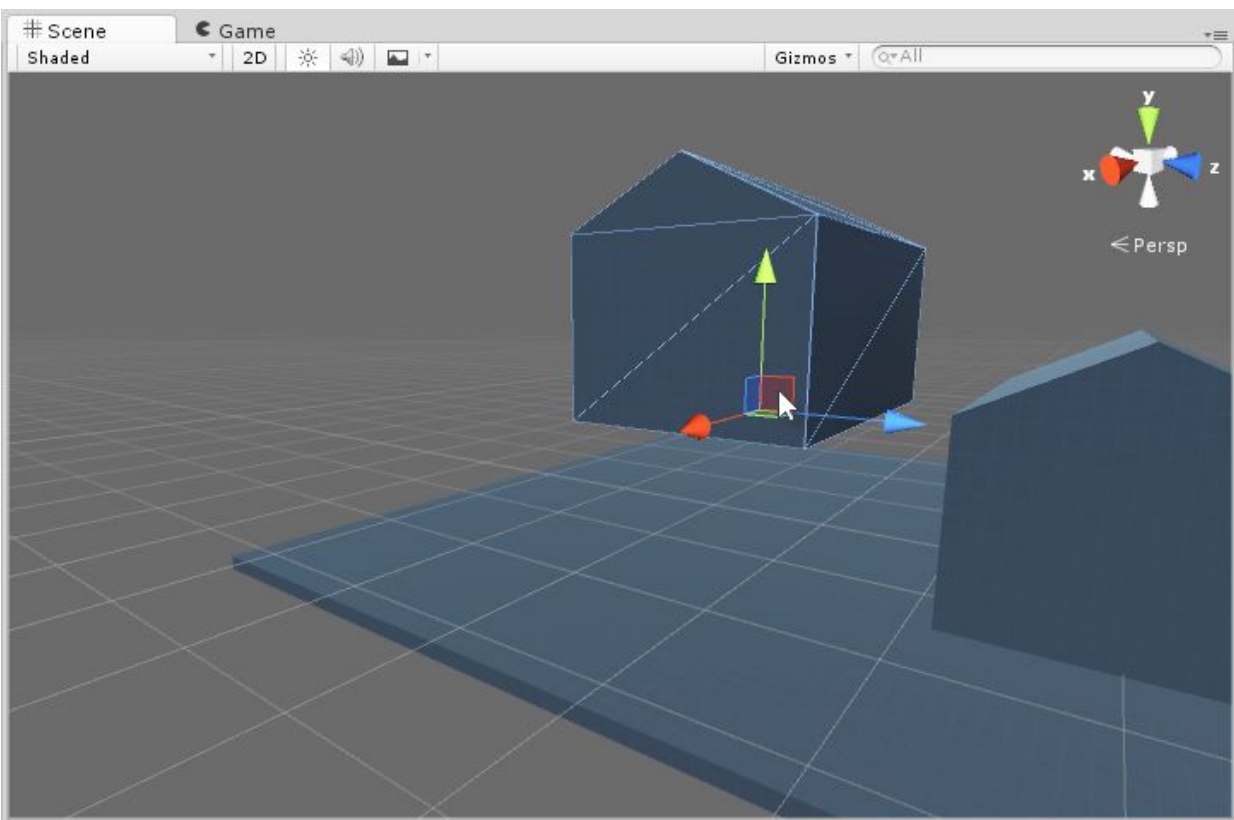


图1.31 使用顶点捕捉（Vertex Snapping）来对没有对齐的物体进行调整

接下来，激活变换工具（W），并且按下键盘上的“V”键，同时移动光标，观察选中网格中最近顶点的标识（Gizmo）光标，如图1.32所示。Unity会要求选择一个进行捕捉操作的源顶点。

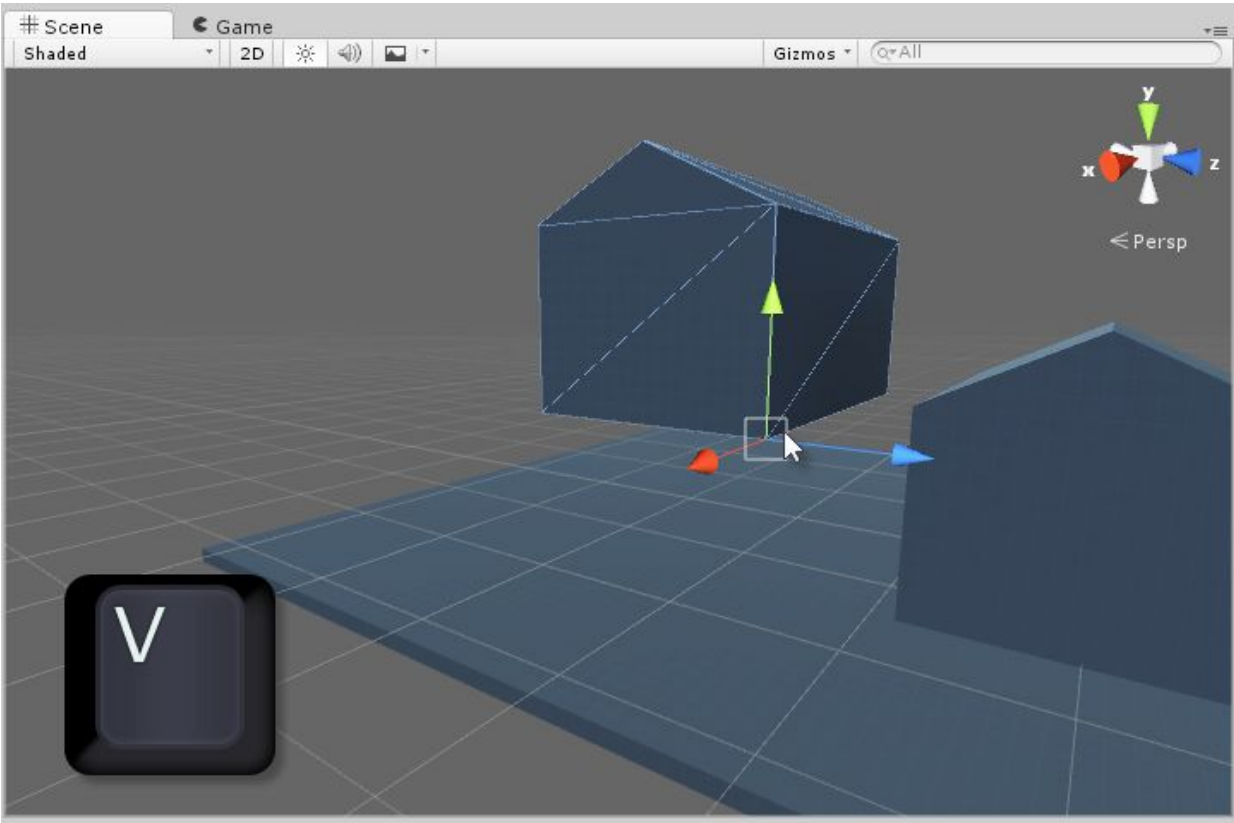


图1.32 按下“V”键来激活顶点捕捉（Vertex Snapping）

按下键盘上的“V”键，同时把光标移动到房屋底部角的位置，然后单击并从角拖动到floor网格的角。然后房屋将会完成与floor网格角对角的顶点捕捉对齐，当完成了这种对齐之后，释放“V”键，此时两个网格的顶点已经对齐了，如图1.33所示。

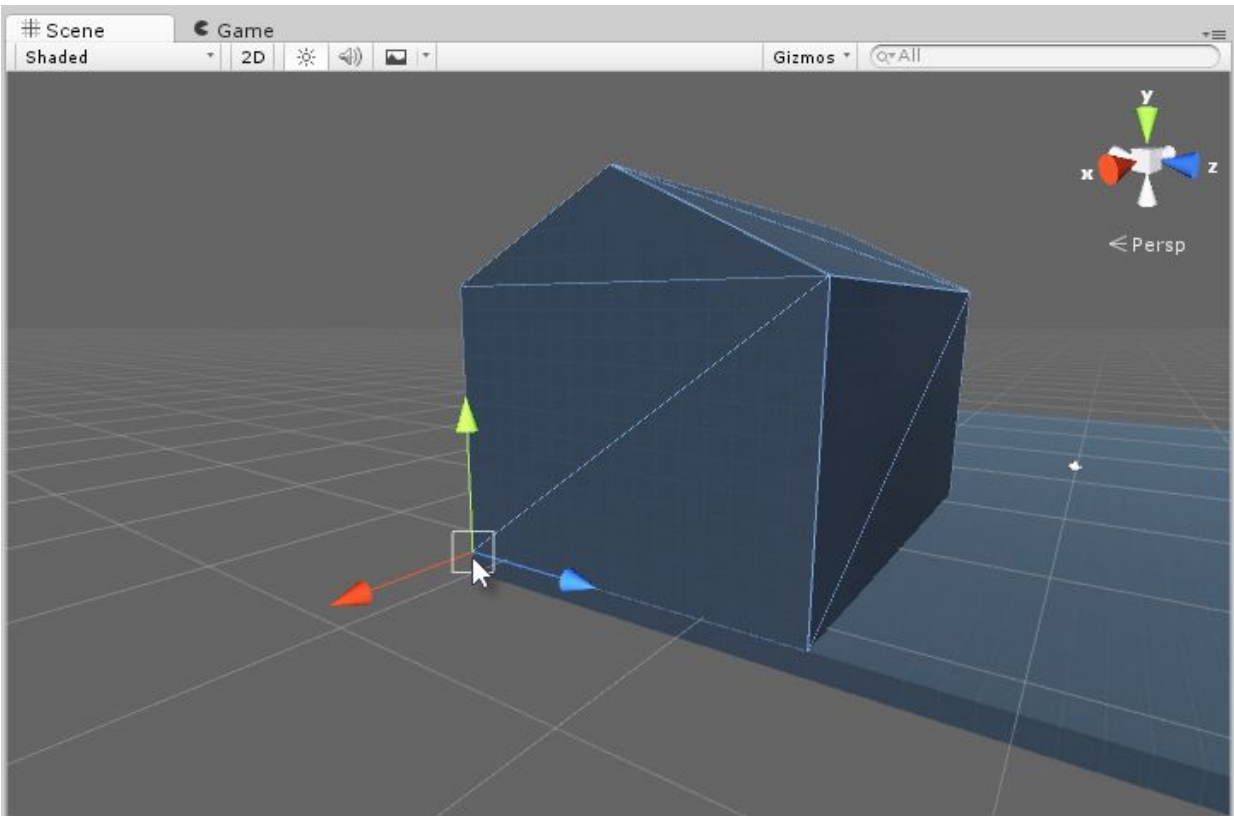


图1.33 通过顶点对齐两个网格

现在，可以使用原型包（**Prototyping Package**）中的网格资源来搭建一个完整的场景了。通过向场景中拖放一些道具，并使用变换、旋转以及缩放操作，就可以实现对这些游戏对象的定位、排放和转动。使用顶点捕捉，可以按自己所愿将所有游戏对象进行对齐。多练习一些这种操作，使用这些工具和资源可以完成图1.34所示的场景安排。



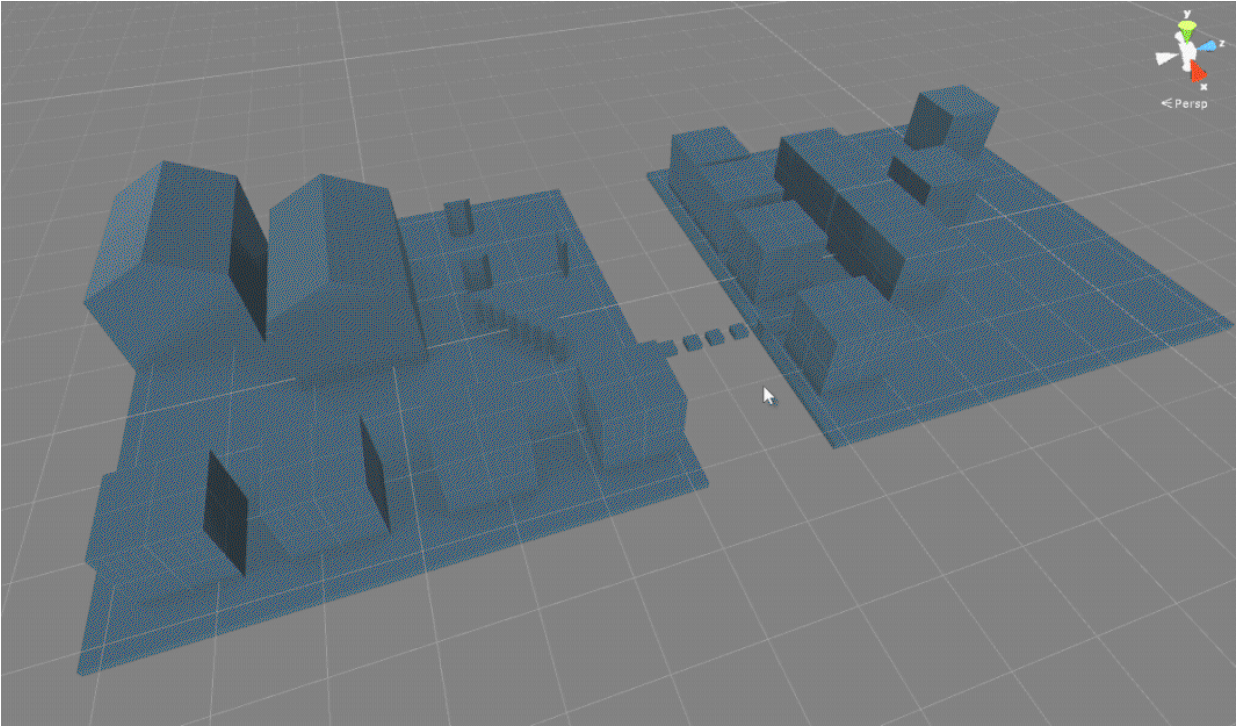


图1.34 构建一个完整的关卡

## 1.7 光源和天空

现在已经完成关卡中基本建筑的模型导入和布局摆放了，只使用很少的几个网格资源和一些基本工具就完成了这些操作。不过，这些工具的功能却是相当强大的，将这些工具组合操作，可以让游戏世界变得丰富多彩，甚至以假乱真。不过，这里还遗漏了一点很重要的事情，那就是光。仔细观察图1.34，这里所有的物体看起来都是单调的，没有光亮、阴影。这是因为场景中并没有配置合适的照明系统，虽然在游戏创建的时候已经默认自带了一个光源对象（**Light**），但是这个光源对象（**Light**）现在并没有起什么作用。



现在这个场景中还没有天空，接下来就为这个金币采集游戏添加上一个天空（Sky）效果，单击场景（Scene）视图顶部工具栏的下拉菜单，然后从下拉菜单中选中“skybox”以启用天空效果。这里的天空盒（Skybox）指的就是一个包围了整个场景的大型立方体，这个立方体的每个内部的侧面都有一个贴图（图像），这些贴图连续在一起，看起来就如同现实中的天空一样。这里选用了“skybox”，就在场景（Scene）视图中显示了一个如图1.35所示的默认的天空。

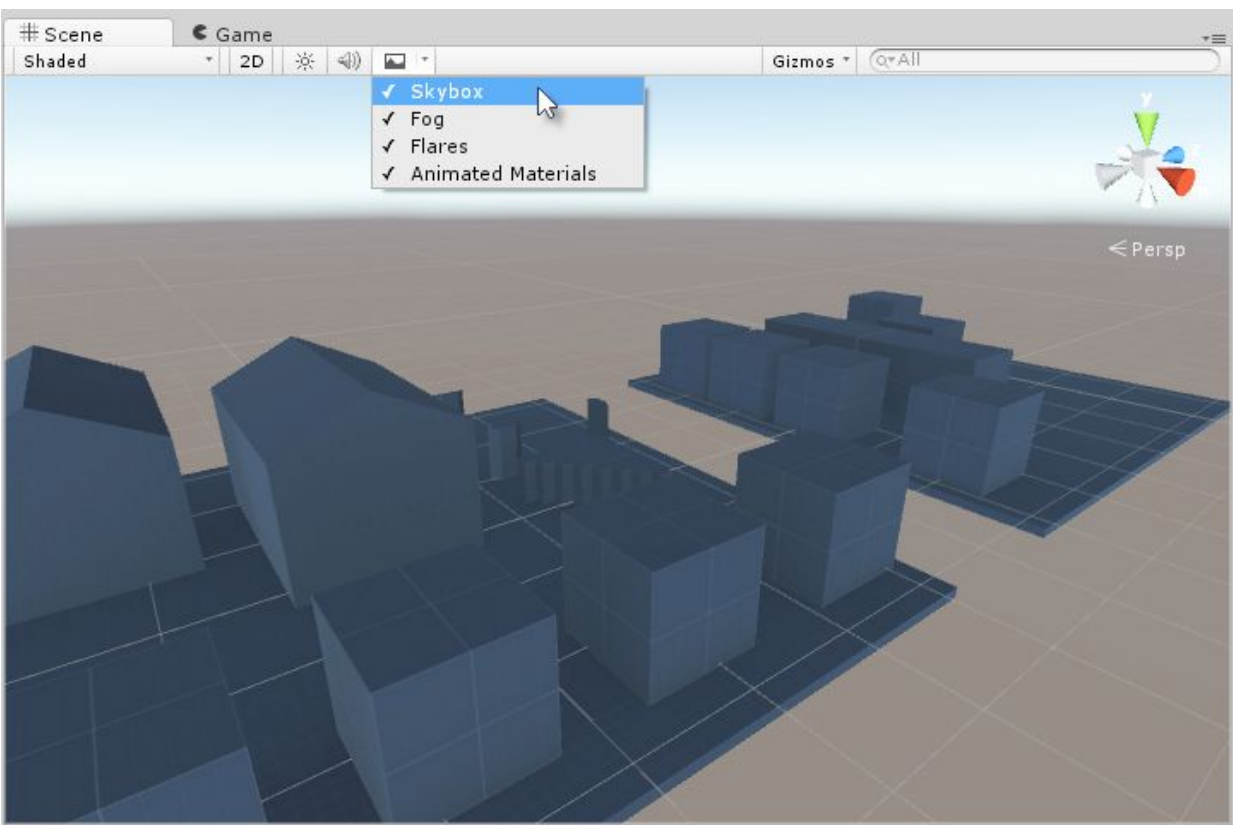


图1.35 启用天空效果

虽然天空效果的启用使得整个场景看起来好多了，但是它却仍然缺乏一个合适的照明系统。场景中的游戏对象缺乏光亮和阴影，为了改善这一点，首先要打开场景（Scene）视图上方（见图1.36）的照明

图标开关，确保照明系统是启用的。注意，这个开关只是决定照明系统是否在场景（**Scene**）视图中起作用，而决定不了在最后的游戏中照明系统是否起作用。

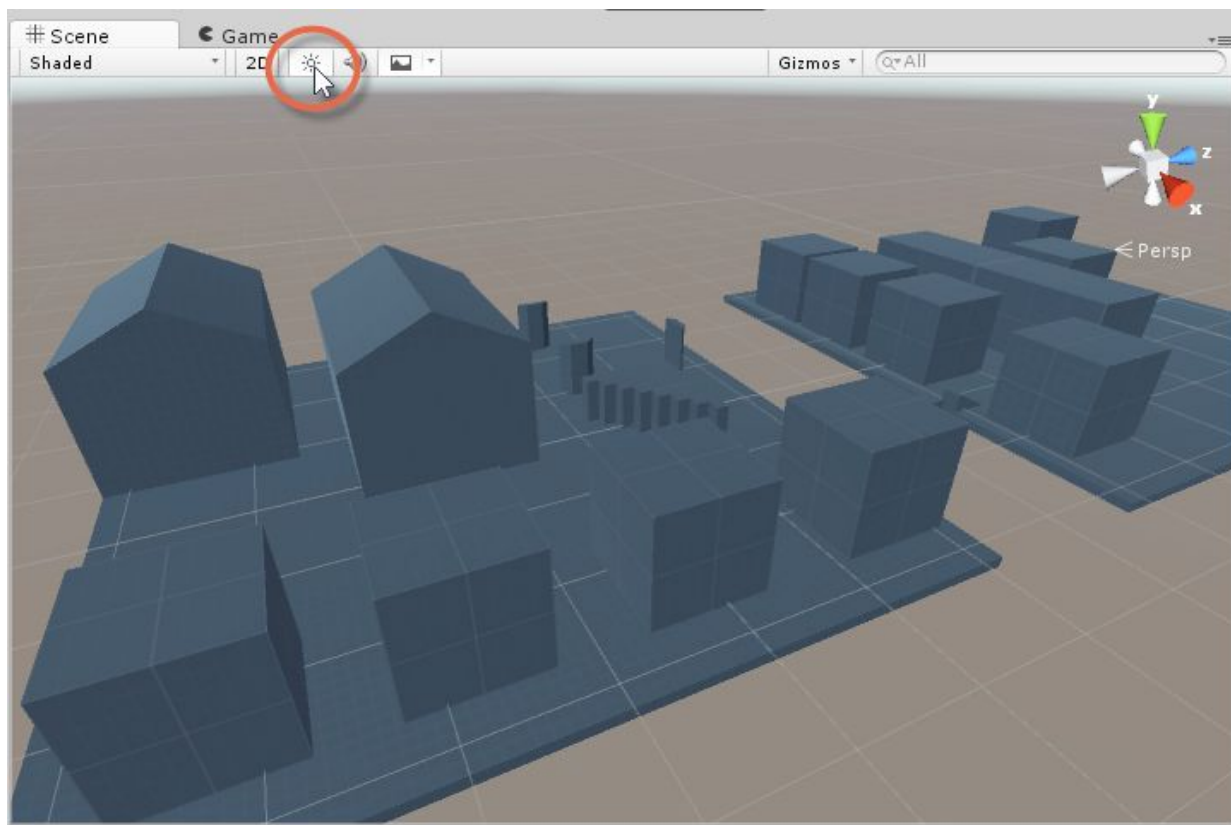


图1.36 在场景（Scene）视图中启用场景照明系统

启用了照明系统之后，就会明显地看到场景与之前有了很大的差异，现在的场景看起来真实多了。也可以选择层次（**Hierarchy**）面板上的“**Directional Light**”来进行旋转操作，以确定场景中的灯光效果。众所周知，一天中的不同时间里，光线的强度和角度都是不同的，现在就可以利用这种操作来实现不同时间中光线的变化。这样将会改变场景的渲染方式，如图1.37所示。

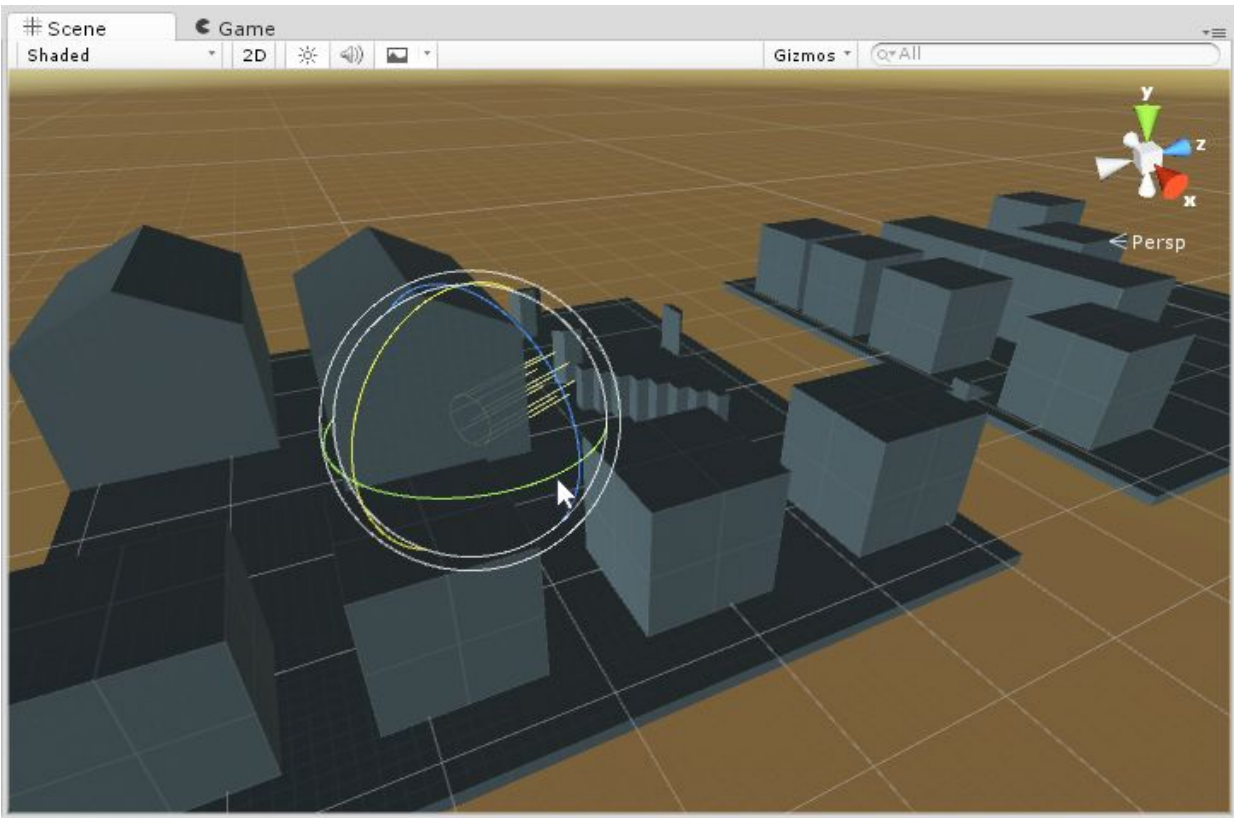


图1.37 转动场景中的方向光（Directional Light）模拟一天中时间的变化

如果要撤销之前对方向光（**Directional Light**）做的任何操作，可以按下键盘上的“**Ctrl + Z**”组合键。为了做好最终及最佳的照明效果，需要把场景中所有不能移动的游戏对象（例如墙壁、地板、桌子、椅子、天花板、草、山、塔等）都标记为静态的。对于**Unity**来说，意味着标记为静态的游戏对象在整个的游戏过程中，无论发生什么，都是永远不会被移动的。通过将游戏对象标记为静态，就可以帮助**Unity**在对游戏渲染和照明处理时进行优化。现在只需要在场景中选择所有不能移动的对象（这包括了目前为止游戏中的所有对象），然后在对象的检查（**Inspector**）面板中选中静态（**Static**）复选框。注意，无需分别地设置静态属性。在为多个对象设置静态属性时，只需要在选择这些对象的，同时按下**Shift**键就能选中所有对象，然后就可以一次性地

在对象的检查（Inspector）面板中为这些对象设置属性，如图1.38所示。

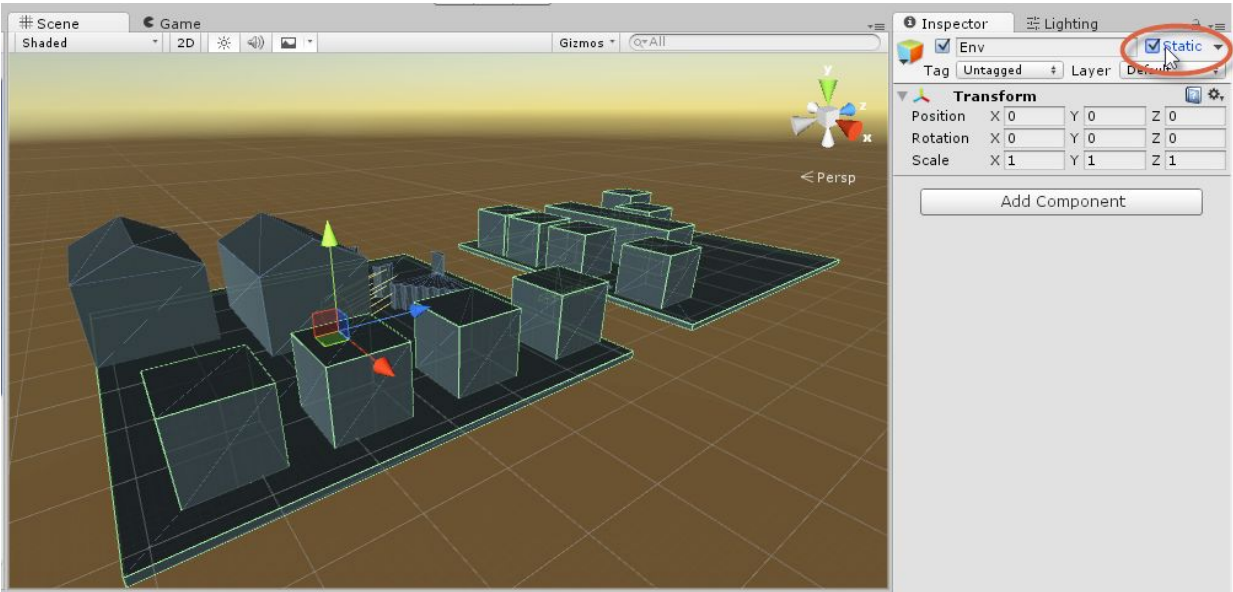


图1.38 为多个不能移动的对象设置静态选项以改善照明和系统性能

当为一个几何图形的游戏对象勾选了静态（Static）复选框之后，Unity将会自动为场景中的光线效果进行计算，这些效果包括阴影、间接照明等。通过计算得到了一些被称为“GI缓存”的数据，这些数据中包含了光的传播路线，向Unity指示了光线应该如何场景中进行传播和反射，才能更真实地模拟现实世界。即便如此，之前勾选了静态复选框的游戏对象还是没有阴影，这将使游戏看起来很不真实。产生这种情况的原因是大部分的网格对象都有一个产生阴影（Cast Shadows）的选项，而这个选项默认是禁用的。现在来消除这个缺陷，首先选中场景中的所有网格对象，然后在检查（Inspector）面板上单击“Mesh Renderer”组件并选中“Cast Shadows”复选框，当完成这个操作后，所有的网格对象就有如图1.39所示的阴影效果了。



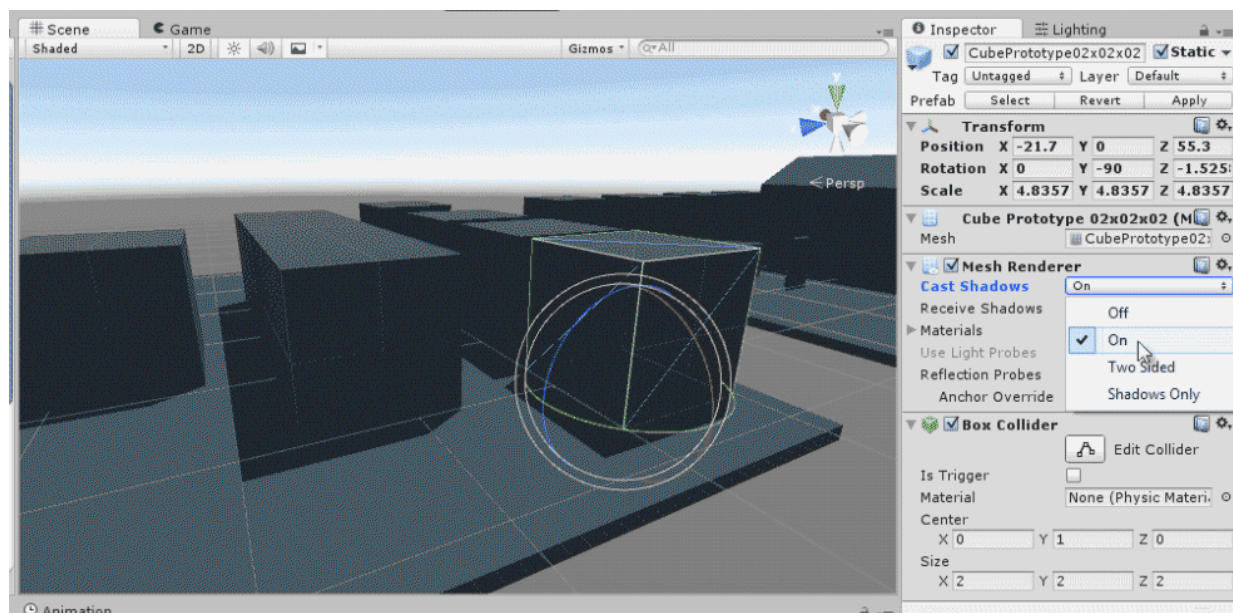


图1.39 从“Mesh Renderer”组件上启用阴影效果

现在的网格对象都已经产生了阴影，下面就此开发过程做个小结。到目前为止，已经创建了一个新的Unity项目，使用网格对象填充了整个游戏场景，而且使用方向光（**Directional Light**）成功地完成了对场景的照明。如果使用第一人称视角模式在这个游戏环境中进行一番游历，那将会是非常棒的体验。好了，我们来看看接下来的操作吧！

## 1.8 游戏测试与游戏选项卡

至此，已经使用Unity自带原型包（**Prototyping Package**）完成了金币采集游戏环境的构建。现在的游戏环境包含了两个主要的岛屿，岛屿上有一些建筑物，两个独立的岛屿通过一个石桥连接在一起，如图1.40所示。可能你的布局与图1.40看起来有一些不同，不过也是非常不错的。

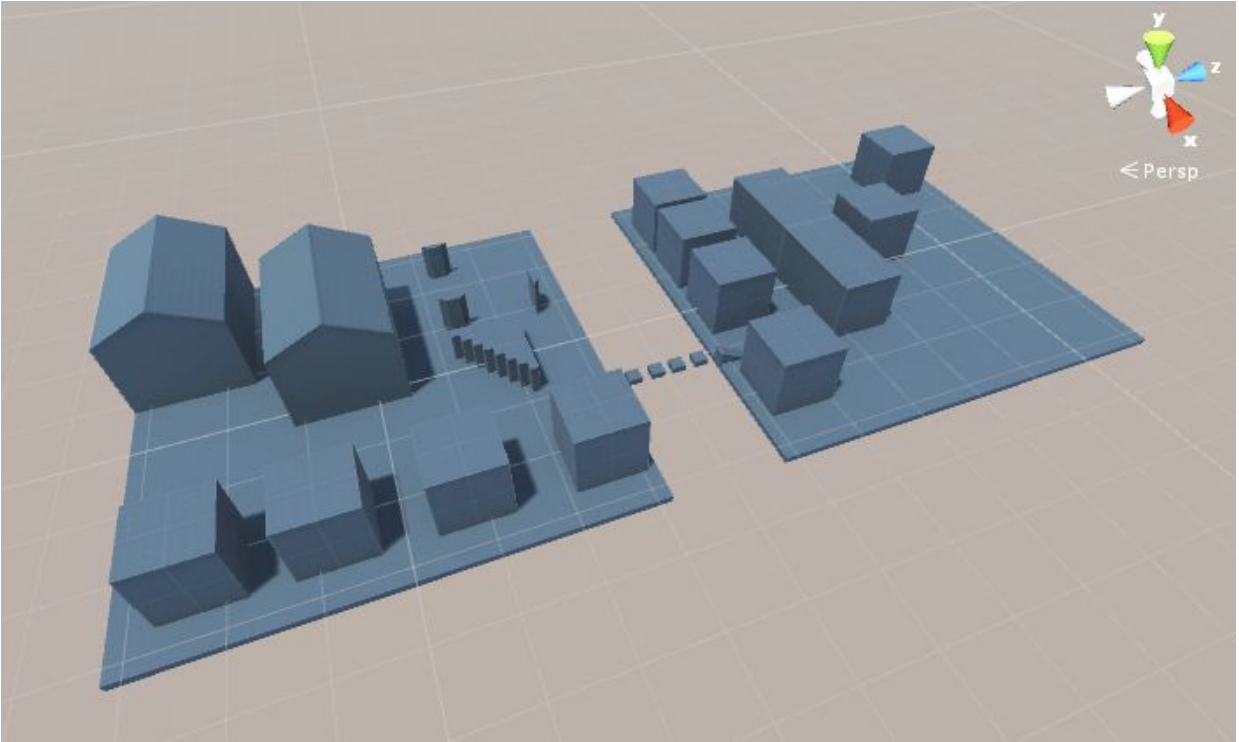


图1.40 目前已经包含了两个岛屿区域的场景

总的来看，场景还是完成得不错，现在需要把这个场景保存一下。可以选择按下键盘上的“Ctrl + S”组合键，或者也可以如图1.41所示，从应用程序菜单上选择“File | Save Scene”选项。如果是第一次对场景进行保存，Unity会弹出一个“Save”对话框，提示为这个场景起一个名字（这里本书将这个场景起名为“level\_\_01”）。



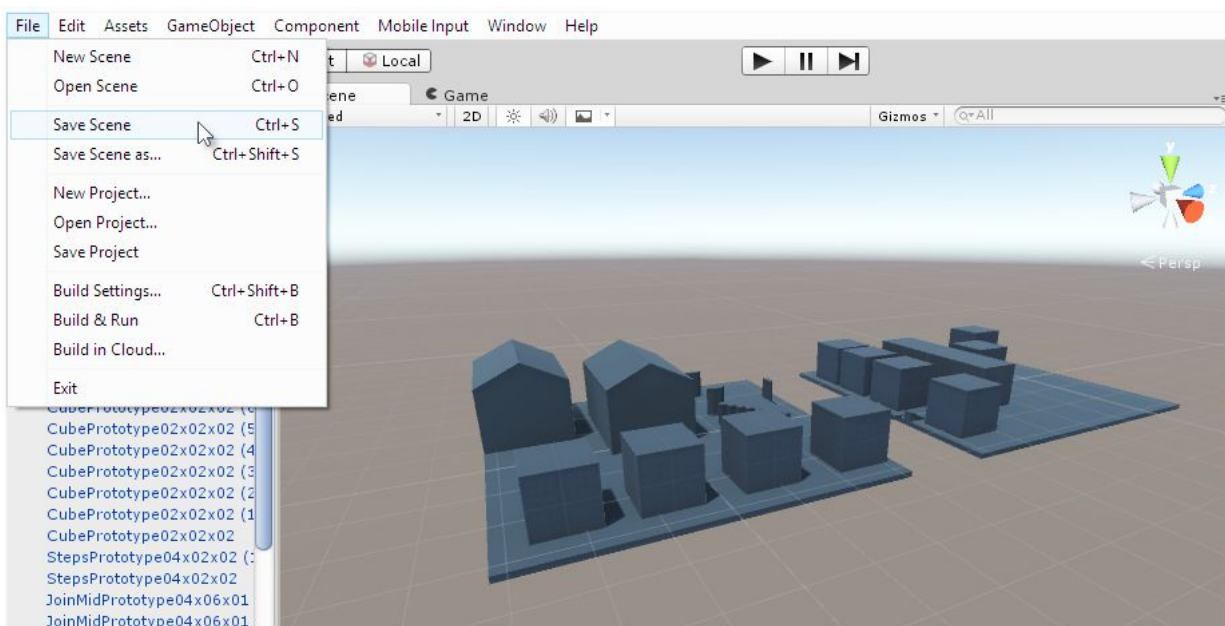


图1.41 对场景进行保存

完成这个场景的保存之后，它就成为了当前项目的场景资源，同时也如图1.42一样出现在了项目（Project）面板中。这意味着当前的场景已经不再像以前是一个临时拼凑在一起的组合了，而是一个真正不可分割的整体。注意，对一个场景进行保存与对一个项目进行保存并不一样，比如在应用程序菜单上就分别有“Save Scene”和“Save Project”两个不同的选项。记住，一个项目是一些文件和文件夹的集合，它包含了游戏中的场景和资源。相比之下，一个场景是项目中的一个资源，外观就像是一个完整的3D地图。这个场景中包含了很多网格、纹理和声音。因此，保存一个项目是指将包括场景在其中的各种文件和资源的配置进行保存。而对场景的保存，只是将指定场景中关卡的变化进行保存。

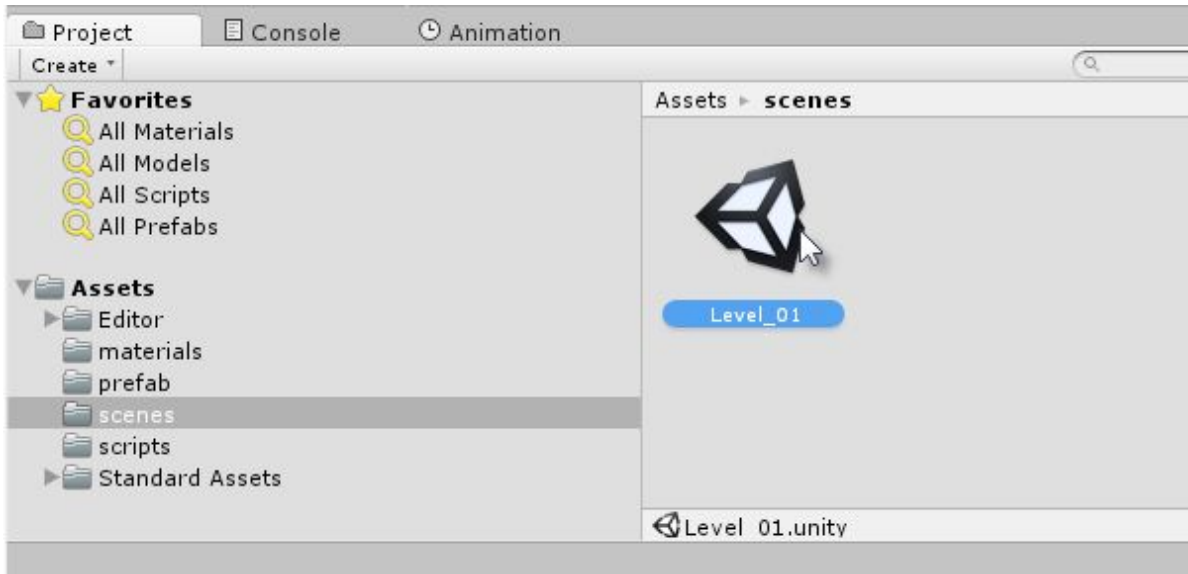


图1.42 保存的scenes被作为资源添加到项目中



从图1.42可以看出，已经将场景保存到了一个名为“scenes”的文件夹中。可以在项目（Project）面板上的任意空白区域单击鼠标右键来为自己的项目创建一个文件夹。另外，依次单击应用程序菜单上的“Assets | Create | Folder”，就可以轻易地使用鼠标拖放来完成对文件夹中的资源进行移动操作。

现在的这个关卡中并没有任何可以玩的东西。它只是一个使用编辑器开发出来的静止的、了无生趣的且与外界毫无交互行为的三维环境。现在给游戏添加可玩性，允许玩家通过控制键盘上的“W”“A”“S”“D”以第一人称视角的方式在这个游戏世界中进行环游和探索。为了实现这个目标，需要向场景中添加一个第一人称视角角色控制器（First-person Character Controller），它是Unity中自带的资源，本身已经包含了实现第一人称视角角色控制的所有功能。依次打

开“Standard Assets | Characters | FirstPerson Character | Prefabs”文件夹，然后从项目（Project）面板拖动“FPSController”资源到场景中，如图1.43所示。

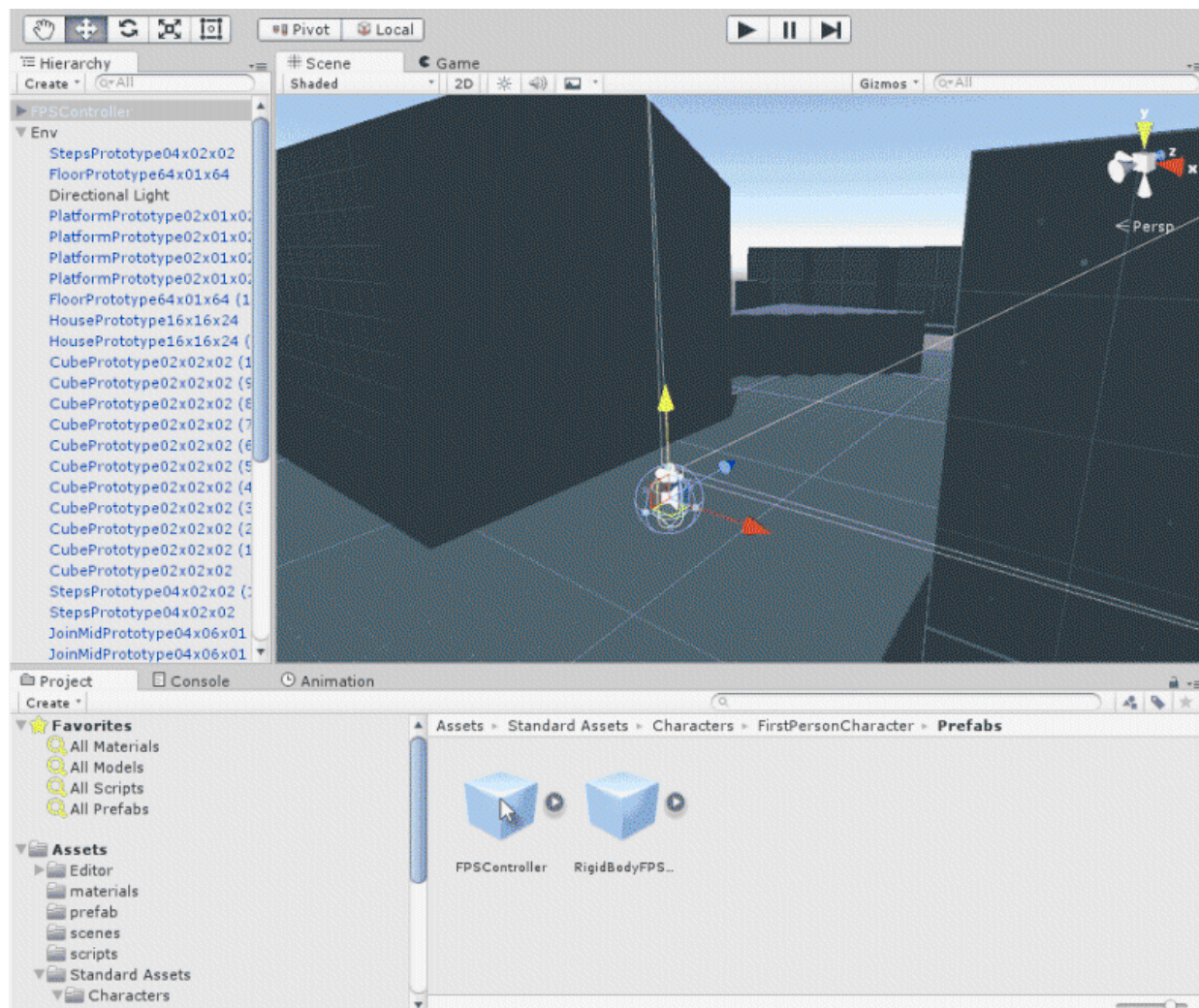


图1.43 向场景中添加第一人称视角角色控制器

当添加了第一人称视角角色控制器以后，就可以单击Unity工具栏上的“Play”按钮以第一人称视角模式来进行游戏，如图1.44所示。



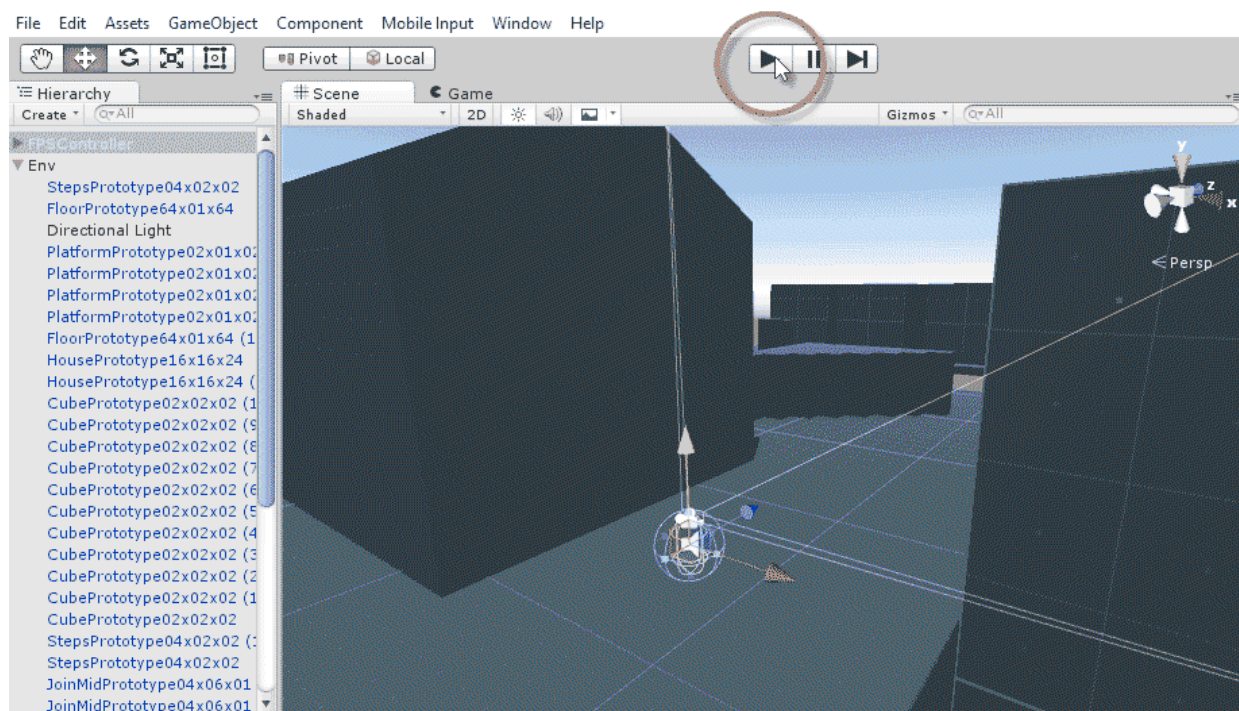


图1.44 单击工具栏上的“Play”按钮对Unity中的场景进行测试

按下“Play”按钮之后，Unity就可以从场景（Scene）选项卡切换到游戏（Game）选项卡，正如所看到的，场景（Scene）选项卡是以一个开发者的视角来观看整个场景的，此时可以对场景进行编辑、制作和设计。相比之下，游戏（Game）选项卡是以一个游戏者的视角来查看整个场景的，只能进行游戏和测试。在游戏者的视角中，只能从主摄像机的角度来看所有的场景。当激活了游戏（Game）选项卡，并启动“Play”模式后，可以使用游戏默认的控制方式开始这个游戏的测试工作。第一人称视角控制器使用键盘上的“W”“A”“S”“D”4个键来控制运动，使用鼠标来控制视角，如图1.45所示。

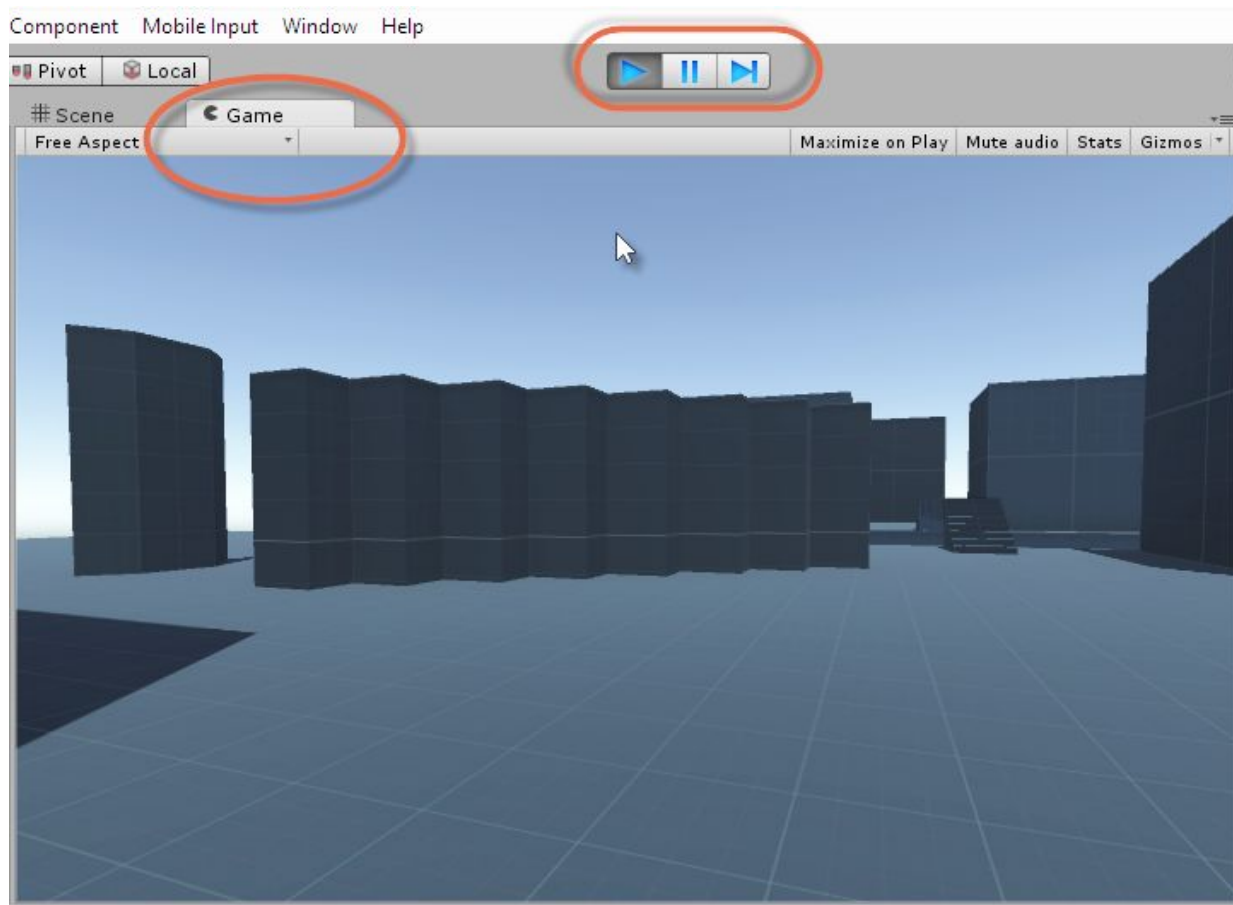


图1.45 在游戏（Game）选项卡中完成对关卡的测试



即使处于“Play”模式中，也可以随时切换到场景（Scene）选项卡中，甚至可以对场景进行编辑和修改，对场景中的对象进行移动和删除。但是要注意的是，所有，在“Play”模式期间进行的改动，在“Play”模式结束之后都会自动复原，也就是这些改动都不会被保留。这是一种特意的设计，它允许在游戏过程中对属性进行编辑，从而可以观察产生的影响，并对问题进行调试，但是却并不真正地对场景进行改动。

现在的关卡中已经可以实现以第一人称视角模式行走了。当结束游戏时，可以再一次单击“Play”按钮，或者按下键盘上的“Ctrl+P”组合键，之后就返回到场景（Scene）选项卡中了。



Unity中还提供了用来暂停和恢复游戏的“Toggle-Pause”按钮。

到现在为止，你应该已经注意到了，当在关卡中以第一人称视角控制器进行游戏的时候，在下面的控制台（Console）窗口中不断地输出一些信息。默认情况下，这个窗口位于Unity编辑器的下方，就在项目（Project）面板旁边，也可以通过在菜单栏上依次选中“Window|Console”人为地打开这个窗口。控制台（Console）窗口会将所有遇到的错误或者警告以消息的形式显示出来。错误都会以红颜色标记，警告都会以黄颜色进行标记，而普通的信息则会以默认的灰色显示。有时，一条消息只会显示一次，而有时却会显示很多次，如图1.46所示。



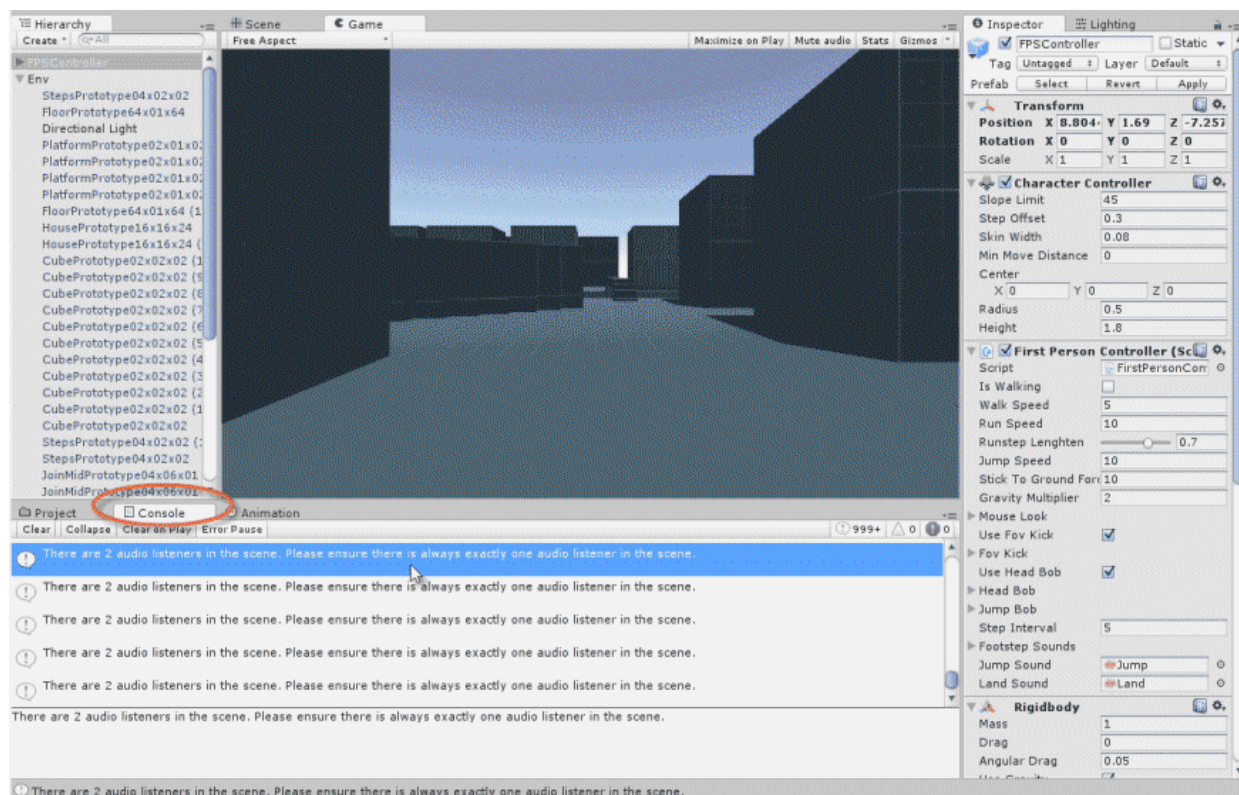


图1.46 控制台（Console）窗口输出信息、警告和错误

正如之前提到过的，在控制台（Console）窗口中可以输出3种不同类型的消息：信息、警告和错误。信息是指基于当前工作的项目给出的最佳建议和意见。警告是指在代码或者场景中出现了一些不太严重的问题，如果不改正，则可能会导致游戏中出现意想不到的行为以及不佳的游戏性能。错误消息描述了需要立即引起注意的代码和场景中的错误。有时，这些错误导致游戏根本无法工作，或者在运行的时候发生错误，从而引起游戏崩溃或者失去响应。此时控制台（Console）窗口就显得十分有用了，因为它可以实现对游戏进行调试。图1.46中就给出了一个关于“Audio Listener”重复的问题。

“Audio Listener”是一个连接到摄像机对象的组件。默认情况下，每一个摄像机都有一个“Audio Listener”组件，它像是一个耳朵，在摄

像机所在的位置接收场景中的各种声音。但是，Unity中并不支持在同一场景中使用多个处于激活状态的“Audio Listener”。这也就意味着在同一地点同一时间只能听到一个声音。那么问题就产生了，因为现在的场景中已经包含了两个摄像机，其中一个是在创建场景时自动产生的，另外一个添加的第一人称视角控制器中自带的。若想确认这一点，只需要在层次（Hierarchy）面板上选中第一人称视角控制器对象，之后单击它名字旁边的三角形图标，这样就会在这个物体的下方显示更多的对象，这些对象都是第一人称视角控制器的组成部分，如图1.47所示。

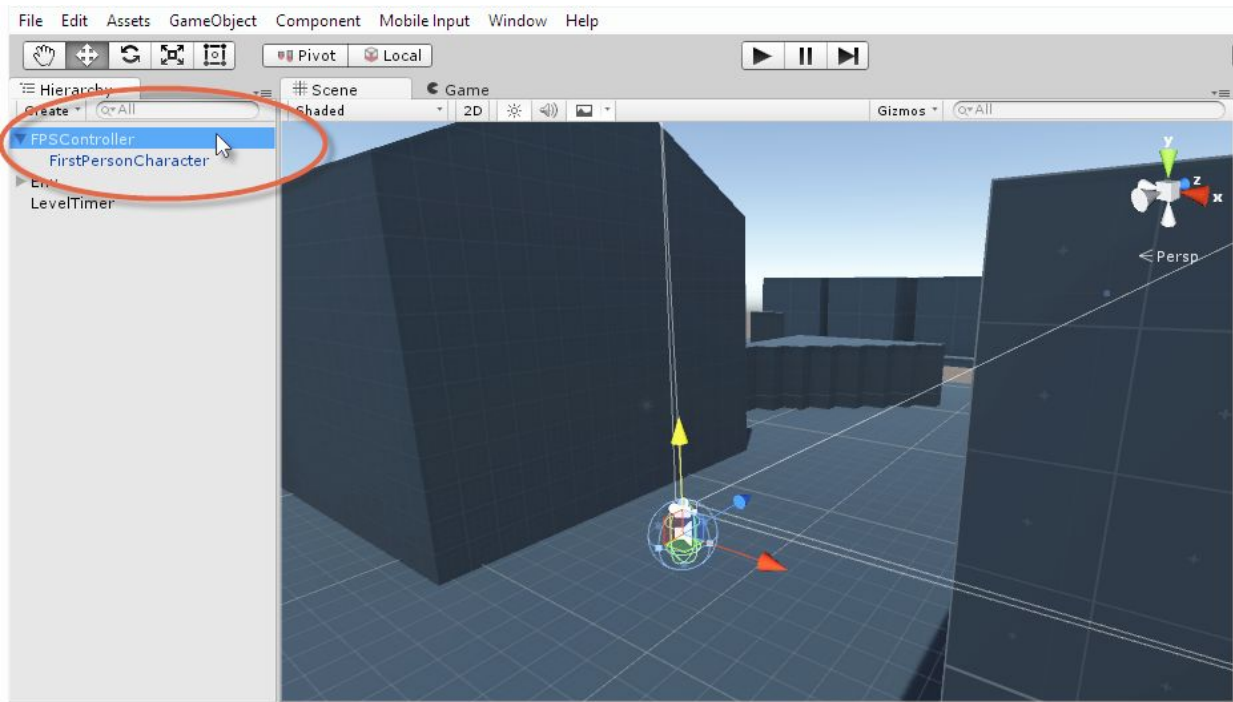


图1.47 找到第一人称视角控制器中的摄像机

选中位于“FPSController”下方展开之后显示的“FirstPersonCharacter”对象，如图1.47所示。“FirstPersonCharacter”对象是“FPSController”对象的一个子对象。这一点从层次（Hierarchy）面

板上的“FPSController”中包含了“FirstPersonCharacter”就可以看出来。子对象会继承它们上一级对象的变换（Transformation）属性。这表示当在场景中对某些对象进行移动或者旋转操作时，所做的变换同样会影响到它的所有子对象。在检查（Inspector）面板中，可以看到这个对象包含一个“Audio Listener”组件，如图1.48所示。

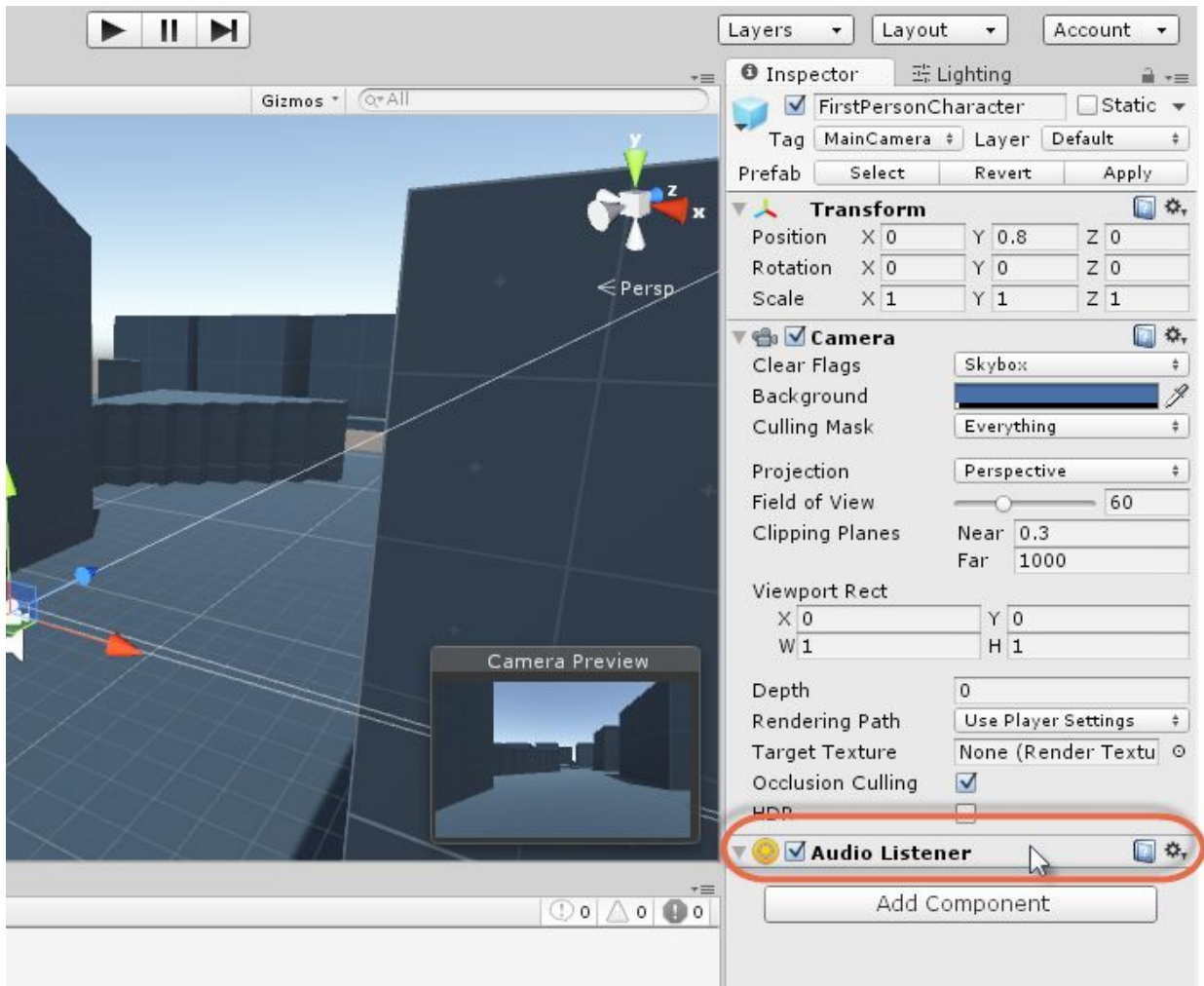


图1.48 包含了“Audio Listener”组件的第一人称视角控制器游戏对象

也可以选择将“Audio Listener”组件从FPSController对象上移除，但是这样做，游戏者在以第一人称视角进行游戏时就再也听不到声音



了。所以，选择将场景创建之初自带的摄像机删除。只需要在层次（Hierarchy）面板上选中图1.49所示的摄像机，然后按下键盘上的“Delete”键即可完成删除操作。这样就可以消除在游戏进行时产生的关于“Audio Listener”的警告。现在可以开始游戏的试玩了！

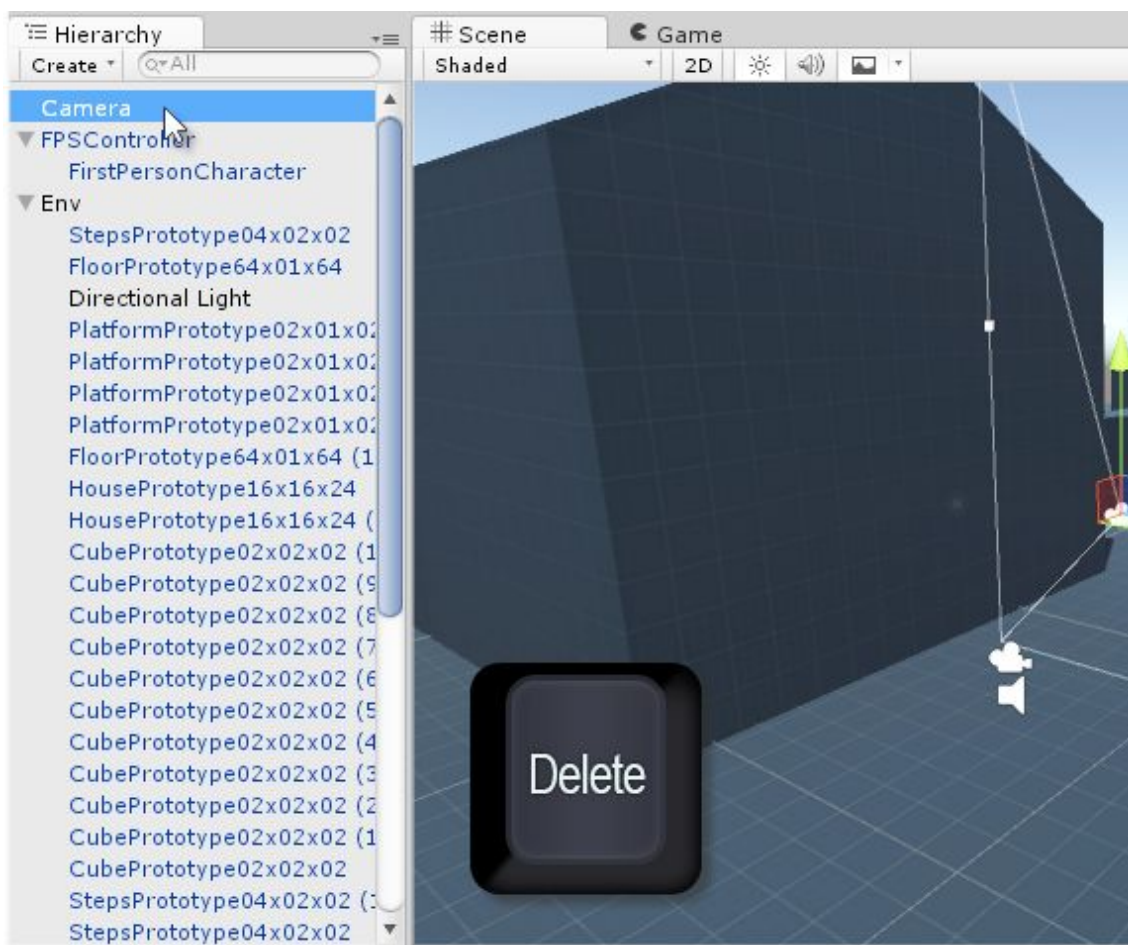


图1.49 删除摄像机对象

## 1.9 添加一个水平面

到目前为止，游戏已经取得了不错的进展，已经可以以第一人称视角的模式在整个游戏环境中行走和探索了。不过，这个游戏环境还

有很多值得改进的地方。例如，**floor**网格现在就如图1.50所示的那样孤零零地悬浮在游戏世界的空中，没有任何的支撑。更离谱的

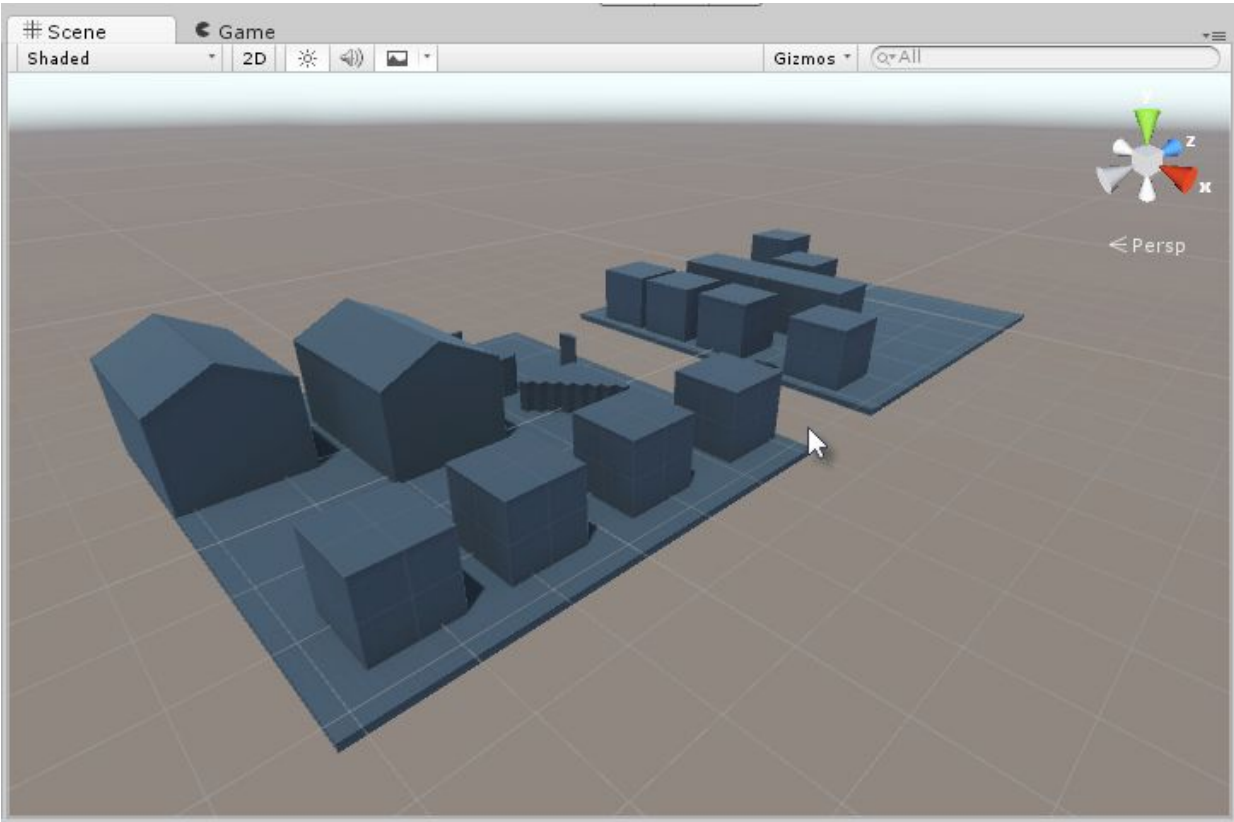


图1.50 整个世界地面悬浮在空中，没有任何的支撑

是，当走出了地面边缘的时候，就会掉下一个无底的深渊。所以现在需要在地面下面添加一片水域，这样场景的环境就更加完整了。

为了向这个游戏中添加水资源，可以使用另一种Unity中预置的资源。在项目（**Project**）面板中依次打开“**Standard Assets | Environment | Water | Water | Prefabs**”文件夹，然后将“**WaterProDaytime**”资源从项目（**Project**）面板中拖动到场景中，如图1.51所示，这是一个圆形的对象，但是实际上的尺寸比所需要的小得多。

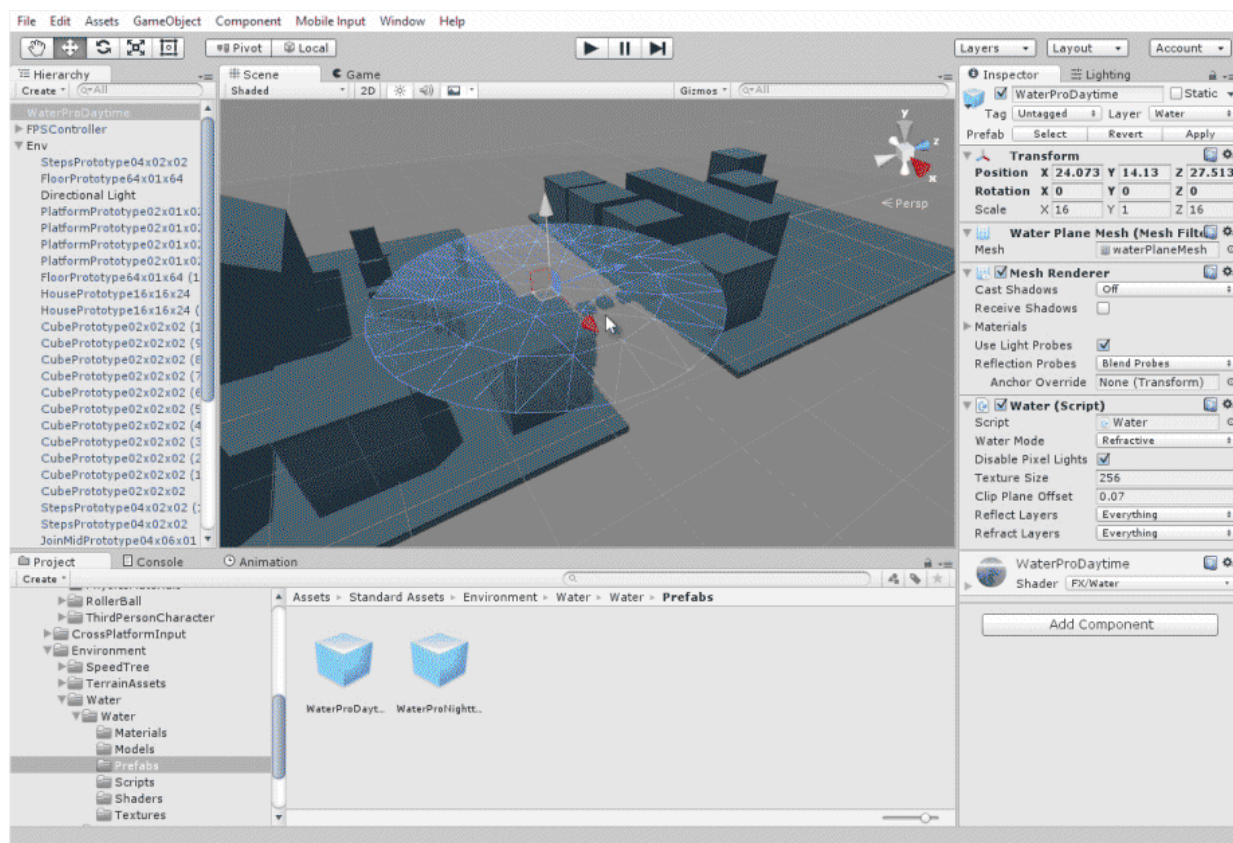


图1.51 向环境中添加水资源

在向场景中添加了水资源预设体（Prefab）之后，将其放置在地平面的下面，然后使用“Scale”工具（R）来变大水资源预设体（Prefab）的平面（X,Z）大小，一直扩大到遥远的地平线为止。这样就使得地面网格看起来好像是一望无际的大海之中的一个小孤岛，完成后的效果如图1.52所示。



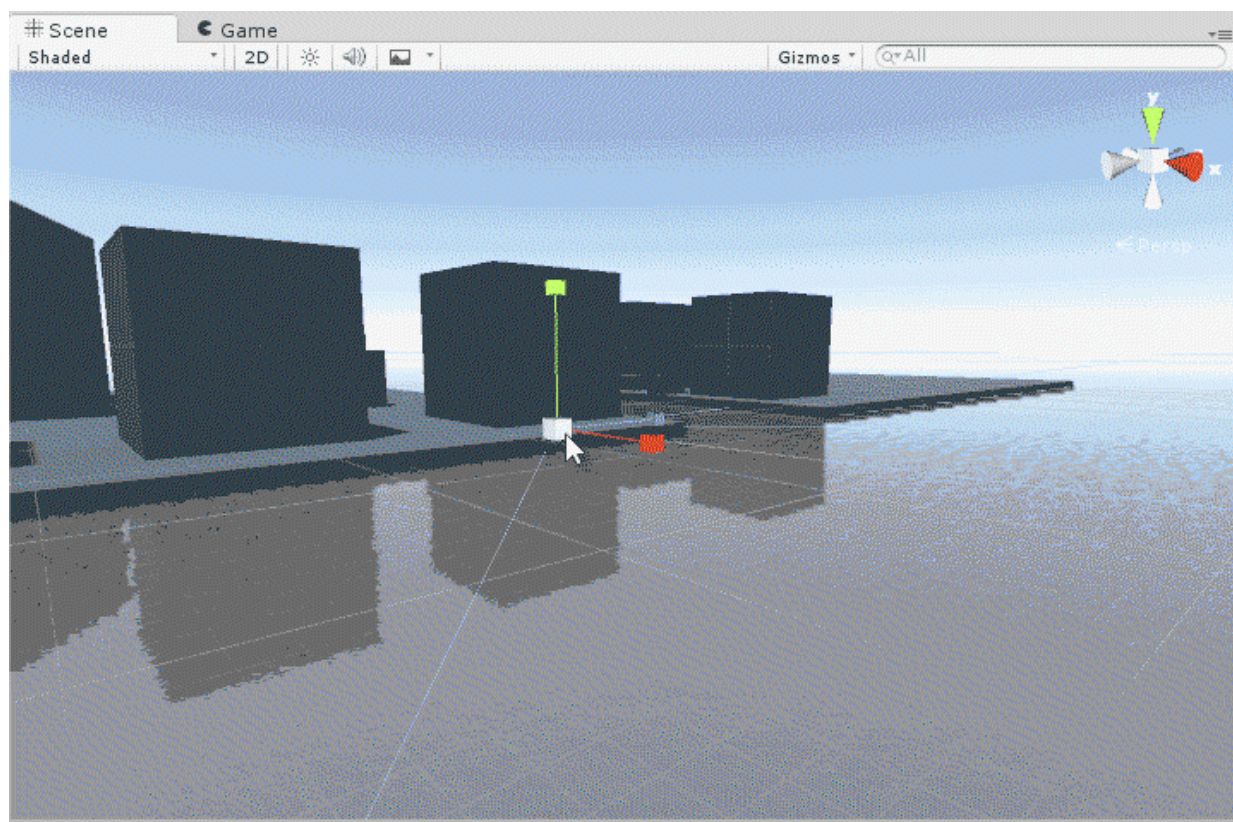


图1.52 环境中水资源面积的调整以及最后的大小

现在，在一个游戏（**Game**）选项卡中开始另一个测试。在工具栏上按下“**Play**”按钮，然后以第一人称视角模式来完成角色的导航。如图1.53所示，可以看到关卡中的水面，当然，在水面上是无法行走的，同样也不能在这里面游泳和潜水。如果真的走到了水面上，就会掉进去，好像这些水根本就不存在一样。现在的水只是一个装饰，仅仅起了一个使场景看起来更真实的作用。

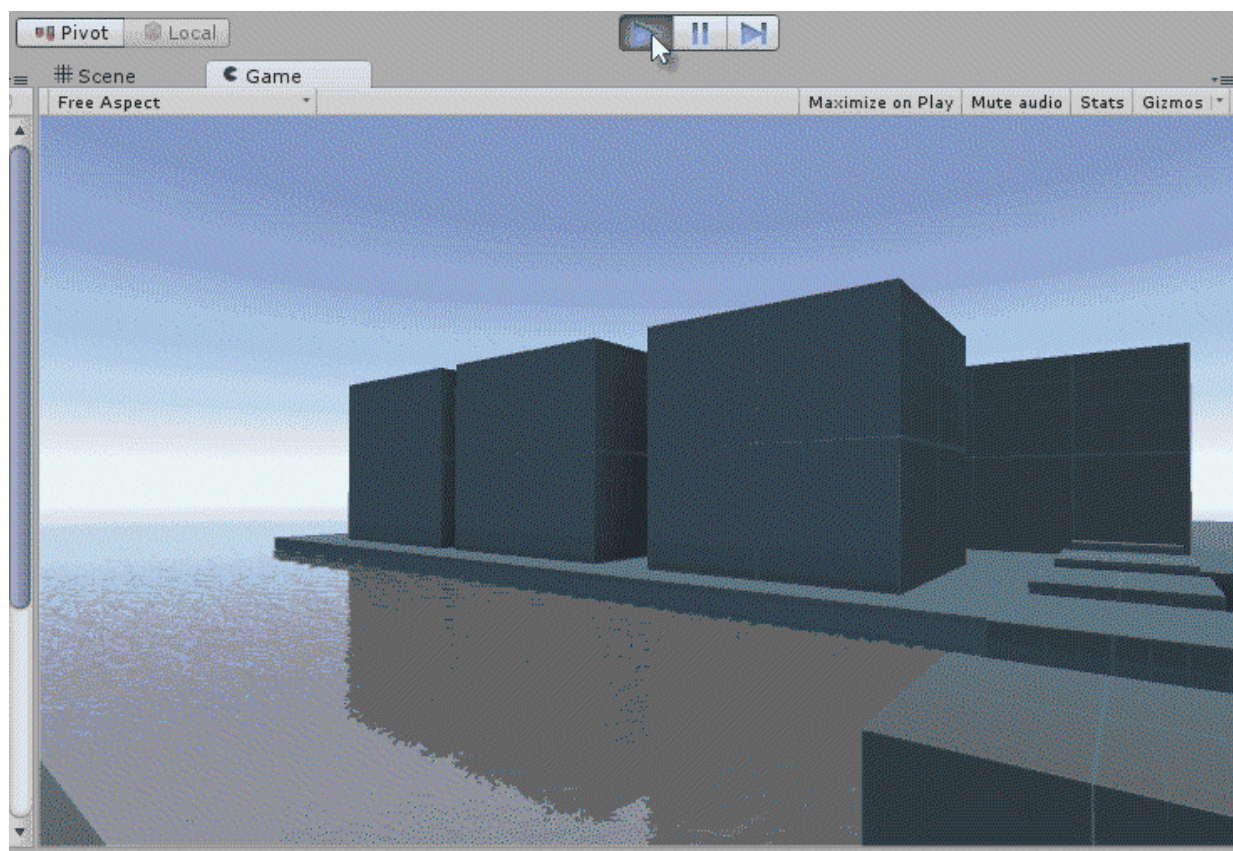


图1.53 以FPS模式对有水的环境进行测试

现在场景中的水是虚无缥缈的，看得见却摸不着，但玩家可以轻易地从这些水中穿过，就好像它们是不存在的一样。Unity现在并不会将这些水当作固态或者半固态的物体来对待。通过给物体添加上一个盒子碰撞体（**Box Collider**）组件，就可以使其具有固体的性质，关于碰撞体（**Collider**）和物理属性（**Physics**）的内容，将会在第3章中进行更详细的讲解。现在，简单快速地给场景中的水对象添加上一个固体的属性，只需要在层次（**Hierarchy**）面板上将水对象选中，然后再在应用程序菜单上依次选中“**Component | Physics | Box Collider**”，如图1.54所示。向一个选中的对象上添加一个组件会改变这个对象本身的表现。即便如此，也千万不要有添加越多组件，对象越完美的错误观



点。其实应该在保证功能的前提下，为一个对象添加尽量少的附件，这种策略可以使 workflow 变得简单、整洁，并具有更好的性能优化。

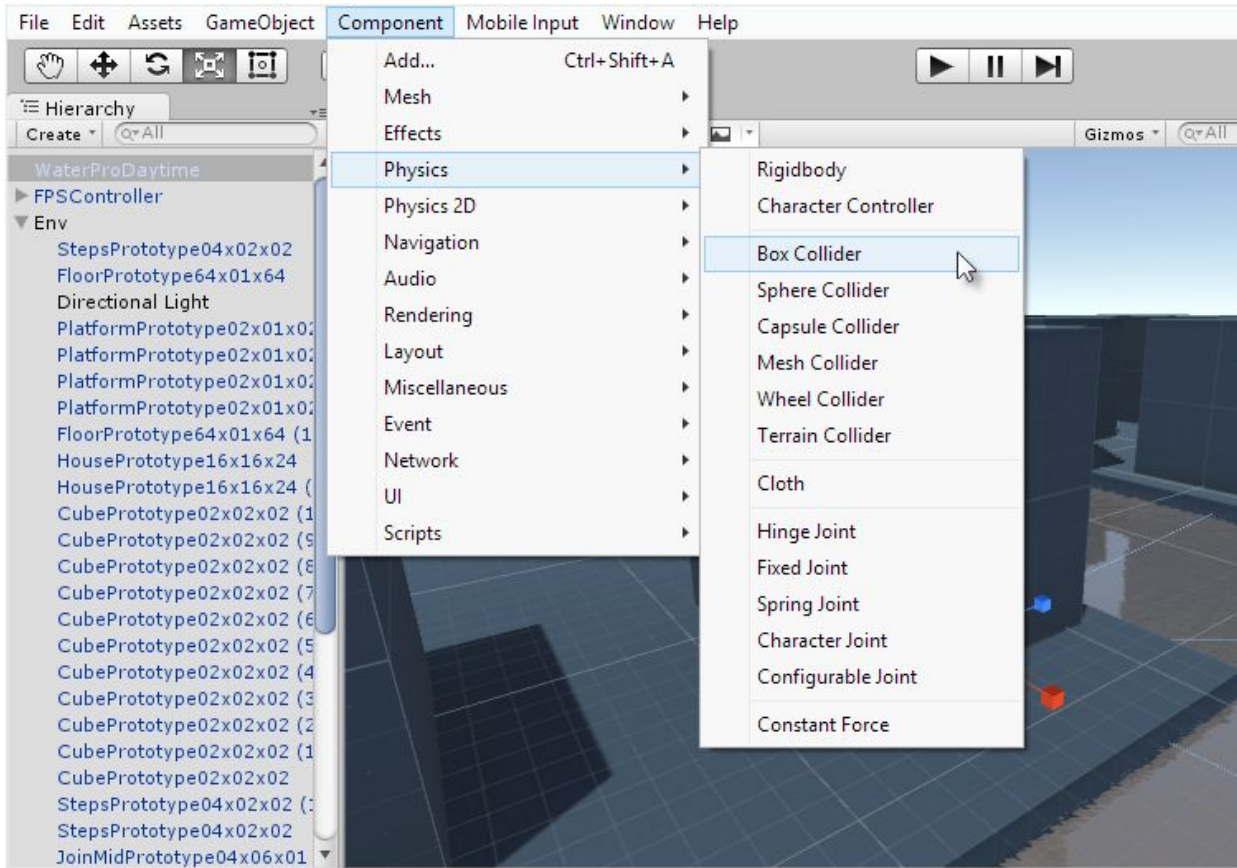


图1.54 向水资源对象添加一个盒子碰撞体（Box Collider）

当向水资源中添加了一个盒子碰撞体（Box Collider）之后，就会在网格周围出现一个绿色的笼状网格。这个绿色的笼状网格大概表示了水资源对象的体积和形状，也就是说Unity认为这个区域就是一个固体，如图1.55所示。

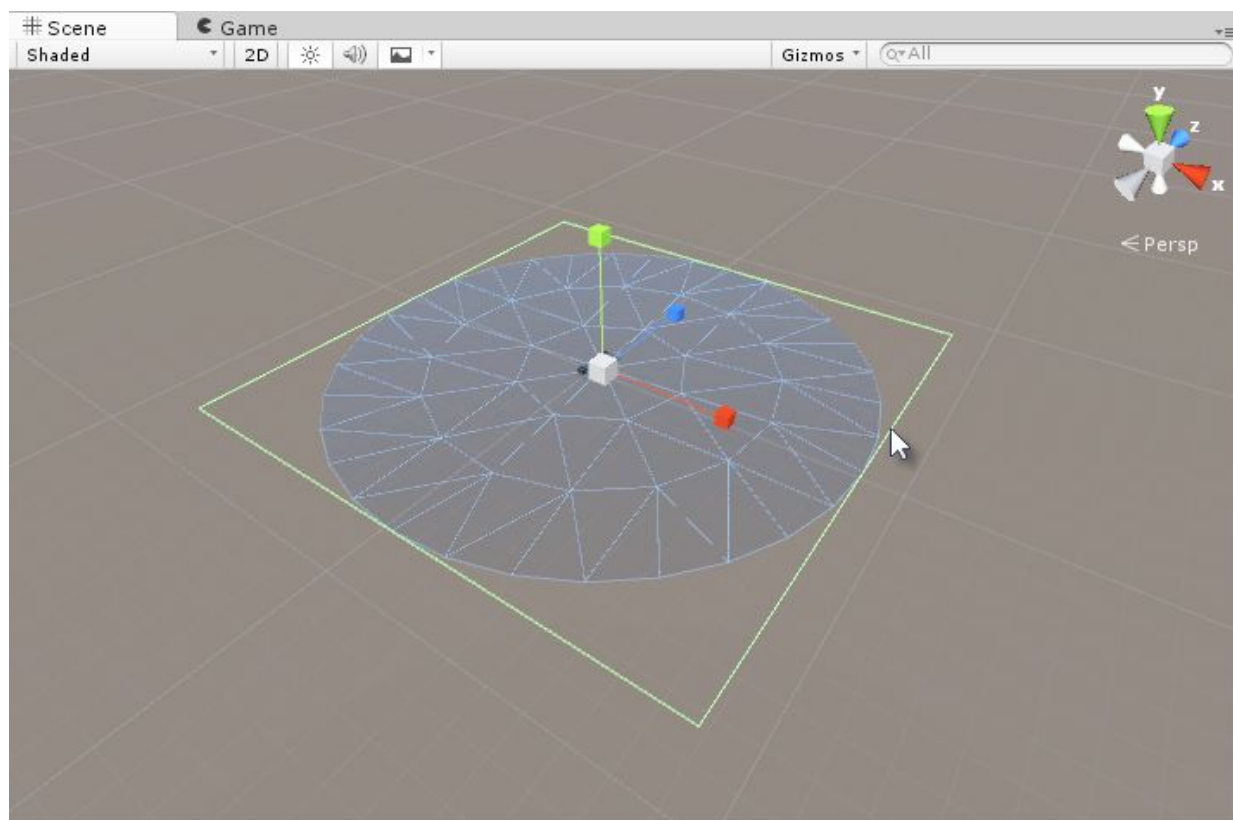


图1.55 盒子碰撞体（Box Collider）的近似体积

现在当控制游戏中的角色走到水面上去时，就会发现角色会在水面上移动，而不会掉下去。当然，更真实的效果应该是这个角色能在水中游泳，虽然现在的“水上漂”有点假，总比掉下去要好多了。想让角色能在水中像真的一样游泳，需要更多的工作，这里暂不涉及。如果需要将盒子碰撞体的功能从水中去掉，返回到初始的虚无状态，则首先选中水对象，然后单击它的“Box Collider”组件右上方的齿轮图标，然后选中下面菜单中的“Remove Component”选项，如图1.56所示。

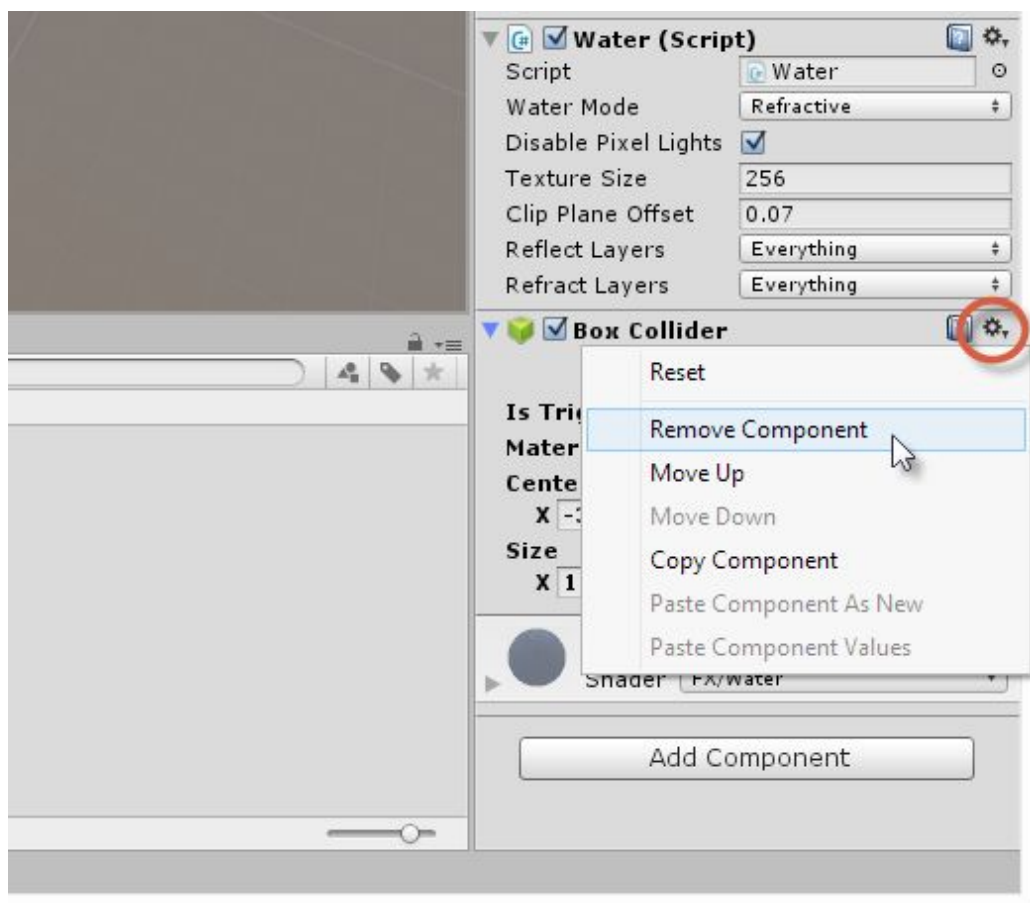


图1.56 移除一个组件

## 1.10 添加一个用来采集的金币

至此，该游戏已经有了很多功能，例如一个完整的环境、一个第一人称视角控制器、一片大海。不过，本章设计的是一个金币采集游戏，但是现在这个游戏场景中还没有任何可以采集的金币。为了实现这些功能，需要编写一些C#脚本，这些脚本要到下一章才会看到。然而可以创建一些金币对象，如将一个圆柱体（Cylinder）对象变形成一个金币。下面先来创建一个圆柱体对象，操作方法是先从应用程序菜单处依次选择“GameObject | 3D Object | Cylinder”，如图1.57所示。

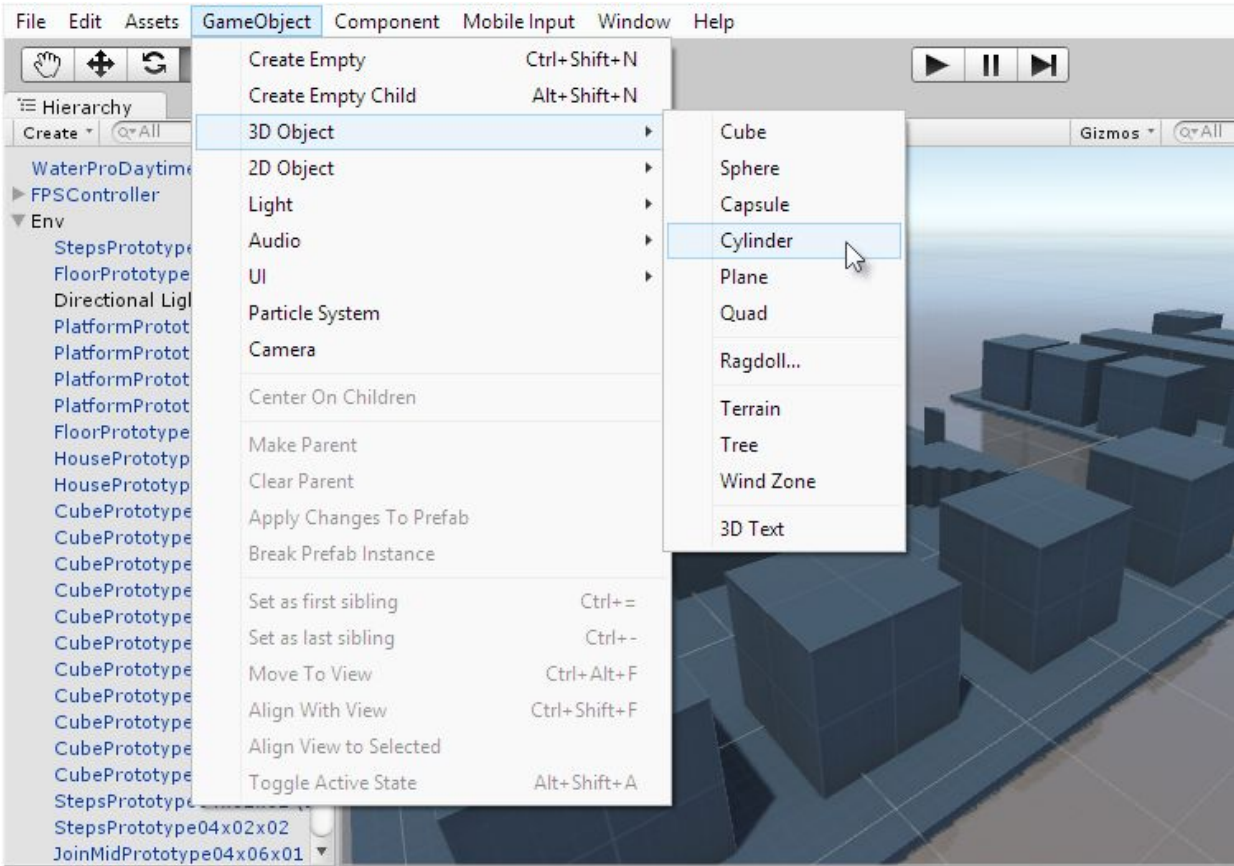


图1.57 创建一个圆柱体（Cylinder）对象

最初，这个圆柱体（Cylinder）对象看起来一点也不像是一个金币。不过它的外形是很容易改变的，只需要按着Z轴将圆柱体（Cylinder）变薄即可。切换到“Scale”工具（R），然后向内减小圆柱体（Cylinder）的长度，如图1.58所示。



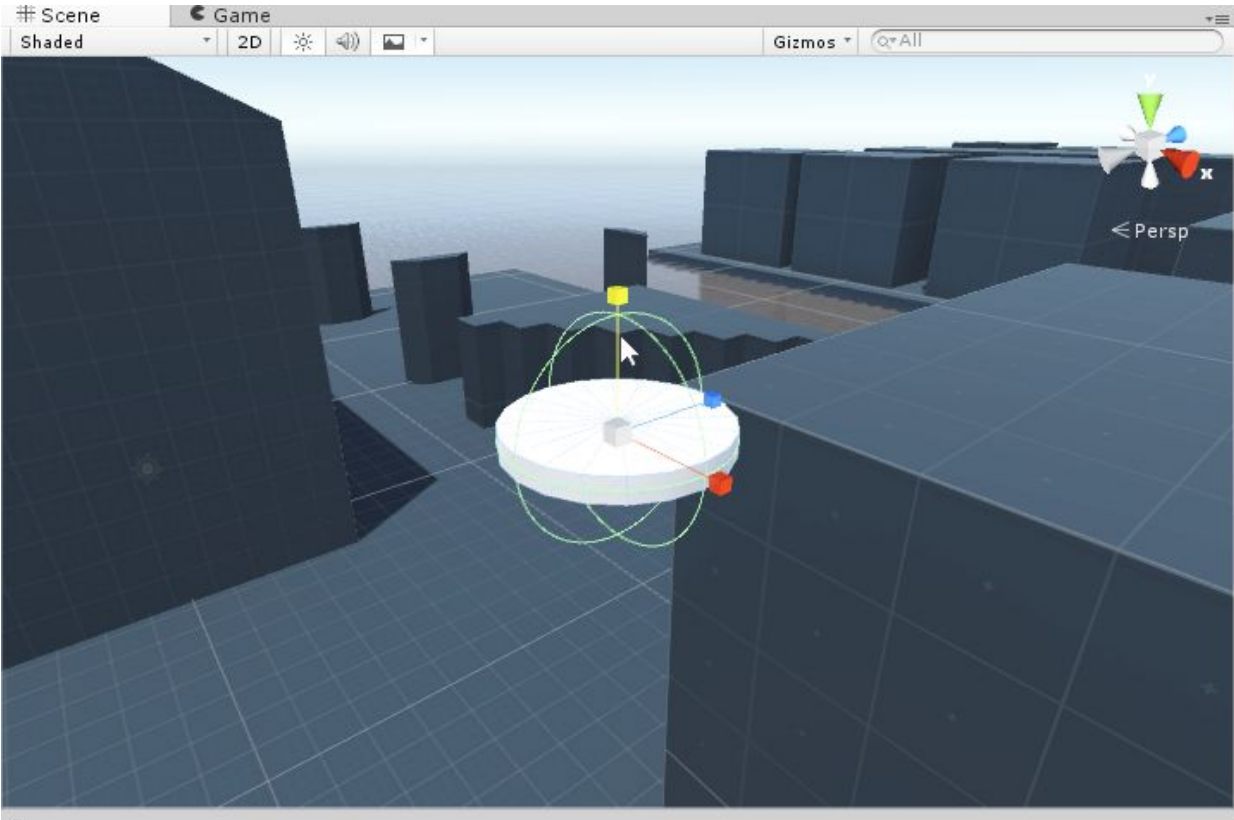


图1.58 将圆柱体（Cylinder）对象变成一个金币形状

在改变了金币的大小和形状之后，它自带的碰撞体的大小与金币的体积就完全不一样了。碰撞体的体积要比金币大很多，如图1.58所示。默认情况下，圆柱体（Cylinder）自带的是一个胶囊型（Capsule）碰撞体，而不是之前见过的盒状碰撞体。可以在金币被选中之后，从检查（Inspector）面板处来改变“Radius”属性的值，这将改变胶囊碰撞体（Capsule Collider）的大小，如图1.59所示。将这个值调小，使得胶囊碰撞体的大小与金币更加接近。或者，选择将胶囊碰撞体（Capsule Collider）完全删除，然后再添加一个盒子碰撞体（Box Collider）来代替。这两种方法都是可行的，可以根据实际情况进行选择。在下一章的代码中将会使用到碰撞体，它将用来检测玩家与金币之间的接触。

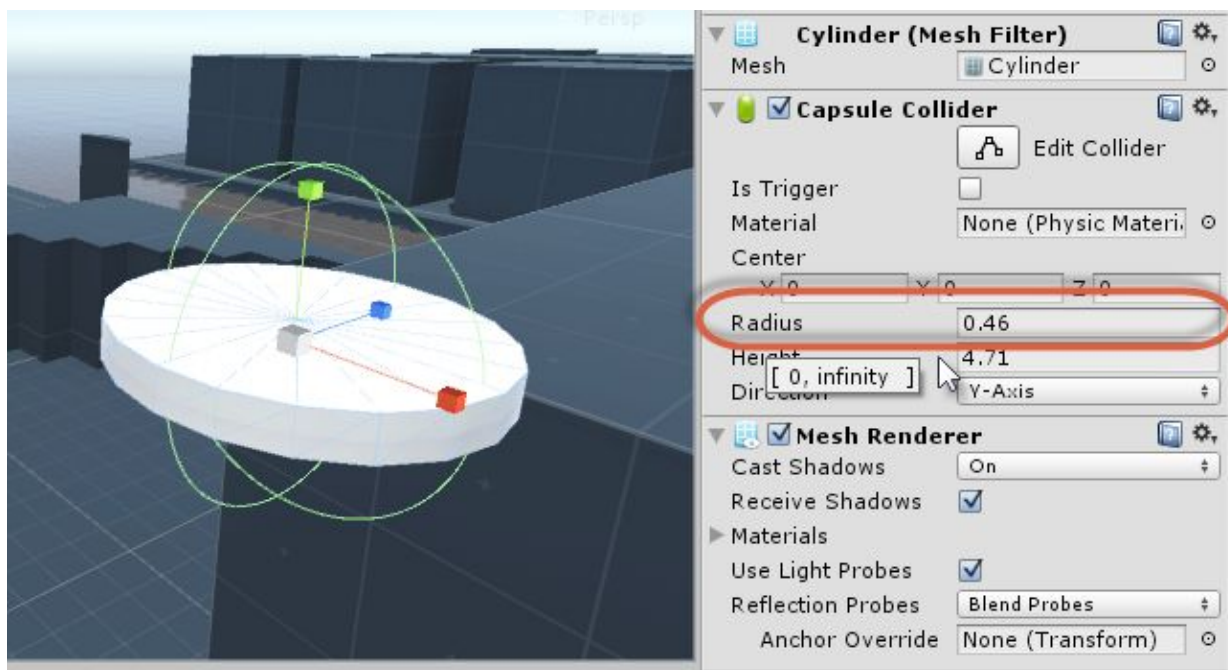


图1.59 调整金币的胶囊碰撞体（Capsule Collider）尺寸

至此，已经将金币的形状和结构设计好了。接下来的一章还要对金币很多方面进行改进。例如，需要给金币添加一个有金属光泽的外观，还要使这个金币可以被采集。本章仅仅使用了Unity自带的基本工具，就产生了一个形状看起来很像金币的对象。

## 1.11 小结

现在已经为金币采集游戏的开发打下了良好的基础，关于这个游戏的功能部分将会在下一章中完成和实现。现在，读者已经掌握了如何从零开始创建一个Unity项目，并使用网格、贴图和场景来充实这个项目。另外，读者也已经了解了如何为游戏创建一个场景，以及如何使用各种资源来完善游戏的功能。这些资源包括水、第一人称视角控

制器和原型资源库等。在下一章中，将为金币添加可采集的特性，并为游戏添加一系列的逻辑，使得游戏有输或者赢的结局。

## 第2章 金币采集游戏（II）

这一章将会在第1章建立好的游戏基础上继续进行。在这个金币采集游戏中，玩家可以第一人称视角模式在整个游戏环境中进行漫游，在游戏规定时间到达之前，寻找并采集到所有的金币。如果在游戏规定时间结束前，玩家就已经完成了所有金币的采集工作，则视为游戏胜利。反之，如果到游戏规定时间结束时，玩家并没有完成所有金币的采集工作，则视为游戏失败。到目前为止，已经在这个项目中添加了一个完整的环境，环境中包括地面、道具、水、一个第一人称视角的控制器，及一个看起来已经具有金币形状但还不能被采集的金币对象。

本章继续完善这个游戏，最后将实现金币可以被玩家采集，系统中还有一个可以检测游戏进行时间的计时器。从本质上来说，这一章中的主要工作是为游戏添加逻辑和规则。为了实现这一目标，需要使用C#编写一些代码，所以本章的学习需要对编程有一些基础。这本书主要是Unity的功能介绍以及如何使用Unity游戏开发引擎，但是程序的编写并不在本书的范围内，因此，假设本书的读者已经有了编写代码的能力。总体而言，本章将涉及的主题如下：

- 材质（Material）的制作
- 预设体（Prefab）
- 使用C#进行游戏编程
- 脚本文件的编写

- 使用粒子（Particle）系统
- 游戏的构建和编译

## 2.1 创建一个金币的材质

在上一章结束时，通过将一个初始的圆柱形（Cylinder）对象进行不均匀的缩放，从而产生了一个基本的金币对象。依次单击应用程序菜单上的“GameObject | 3D Object | Cylinder”可以创建该对象，结果如图2.1所示。从概念上来说，金币对象是游戏逻辑中的一个基本单位，玩家在游戏时间耗尽之前要尽力地去采集这些金币。这意味着这些金币并不能仅仅是被看到就行了，其主要目的是实现游戏的功能，玩家是否收集到了金币将意味不同的游戏结果。因此，现在的金币对象还缺少两个重要的方面。首先，这个金币看起来很阴暗，而且颜色还是灰色的，它完全无法吸引玩家的注意力。再者，这个金币对象无法被玩家采集。不过，玩家可以穿过金币，就好像什么都没有发生一样。

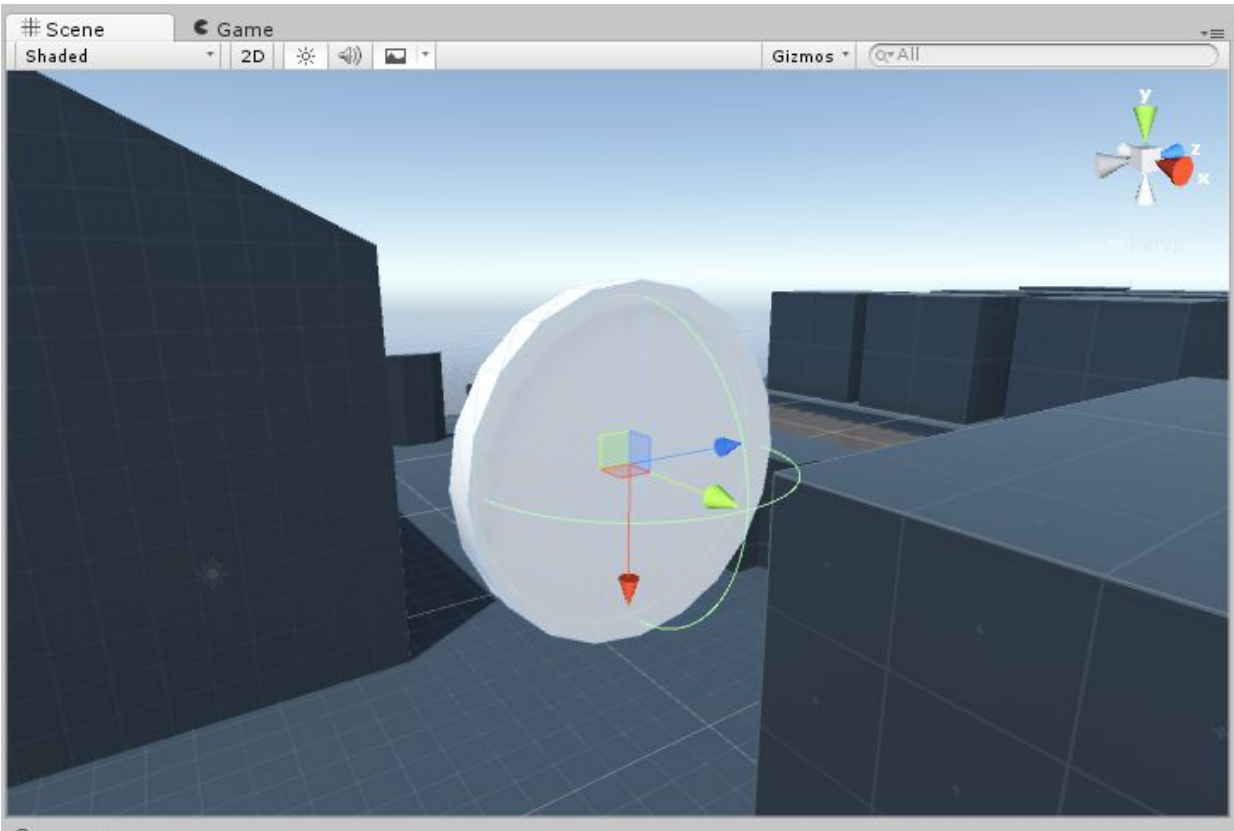


图2.1 到目前为止的金币对象



本章和下一章中讲到的金币采集游戏的完整项目程序可以在本书配套文件的Chapter02/Collection Game文件夹中找到。

这一节将集中讲述如何使用材质（Material）改善金币的外观。一个材质其实就是一个算法（或者说指令集），它指定了金币的外观所呈现的样式。一个材质并不仅仅指物体在颜色方面看起来怎样，它同样定义了物体表面是光滑的还是粗糙的，光照射到物体上之后是发生镜反射还是漫反射。这一点必须要搞清楚才能明白为什么贴图 and 材质



是两种不同的东西。贴图是指一个加载到计算机存储器中的图像文件，可以通过它的UV贴图将一个三维物体覆盖上。相反的，材质定义了一个或者多个贴图是如何结合在一起，并应用到了一个对象外观的塑造上的。如果需要在项目中创建一个新的材质资源，就可以在项目（Project）面板的空白位置单击鼠标右键，然后在弹出的上下文菜单中，选择“Create | Material”，如图2.2所示。或者在应用程序菜单处依次选中“Assets | Create | Material”。

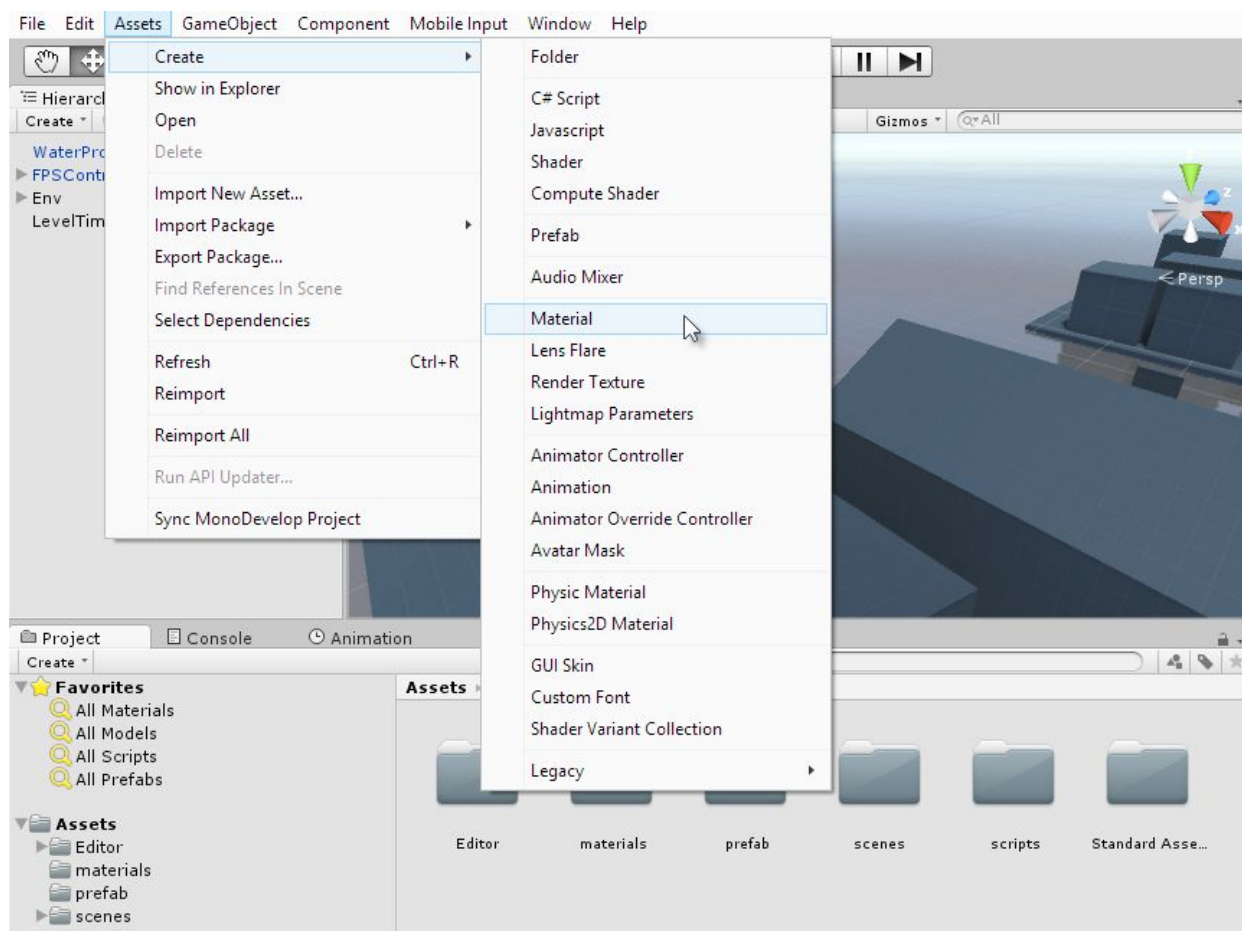


图2.2 创建一个材质



有时一个材质也可以被称作“Shader”。如果需要，就可以使用着色器语言（Shader Language）来创建一个自定义的材质。当然也可以使用Unity中自带的一些插件，例如“Shader Forge”等。

当成功地创建了一个新的材质之后，接下来就要给这个材质起一个名字。此处将这个材质命名为“mat\_GoldCoin”。这个名字的前半部分为“mat”，这样能让大家一下子就知道这是一个材质资源。现在只需要在文本编辑区域输入新的名字，即可完成对这个材质的重命名。以后也可随时在这个材质上双击鼠标，来完成对名字的修改，如图2.3所示。

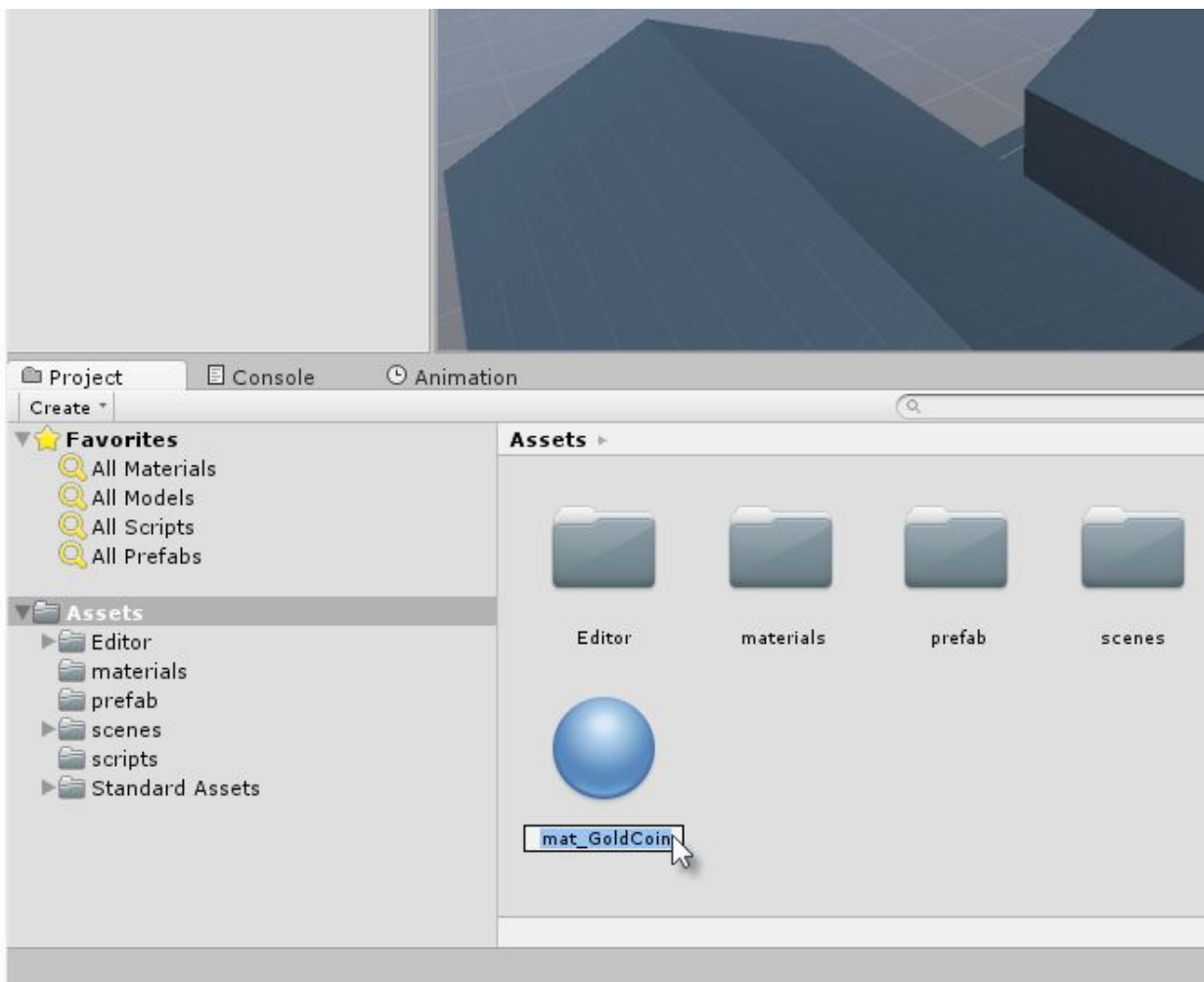


图2.3 为一个材质重命名

接下来，如果还没有选中这个材质资源，那么就在项目（**Project**）面板上选中它，这时其所有属性都会在对象检查（**Inspector**）面板处显示出来。这里的属性数量很多。注意，可以在这个对象的**Inspector**面板底部看到这个材质基于当前设定的预览图，当在检查（**Inspector**）面板处修改了材质设定，下面的预览图也会及时地根据调整进行更新，如图2.4所示。

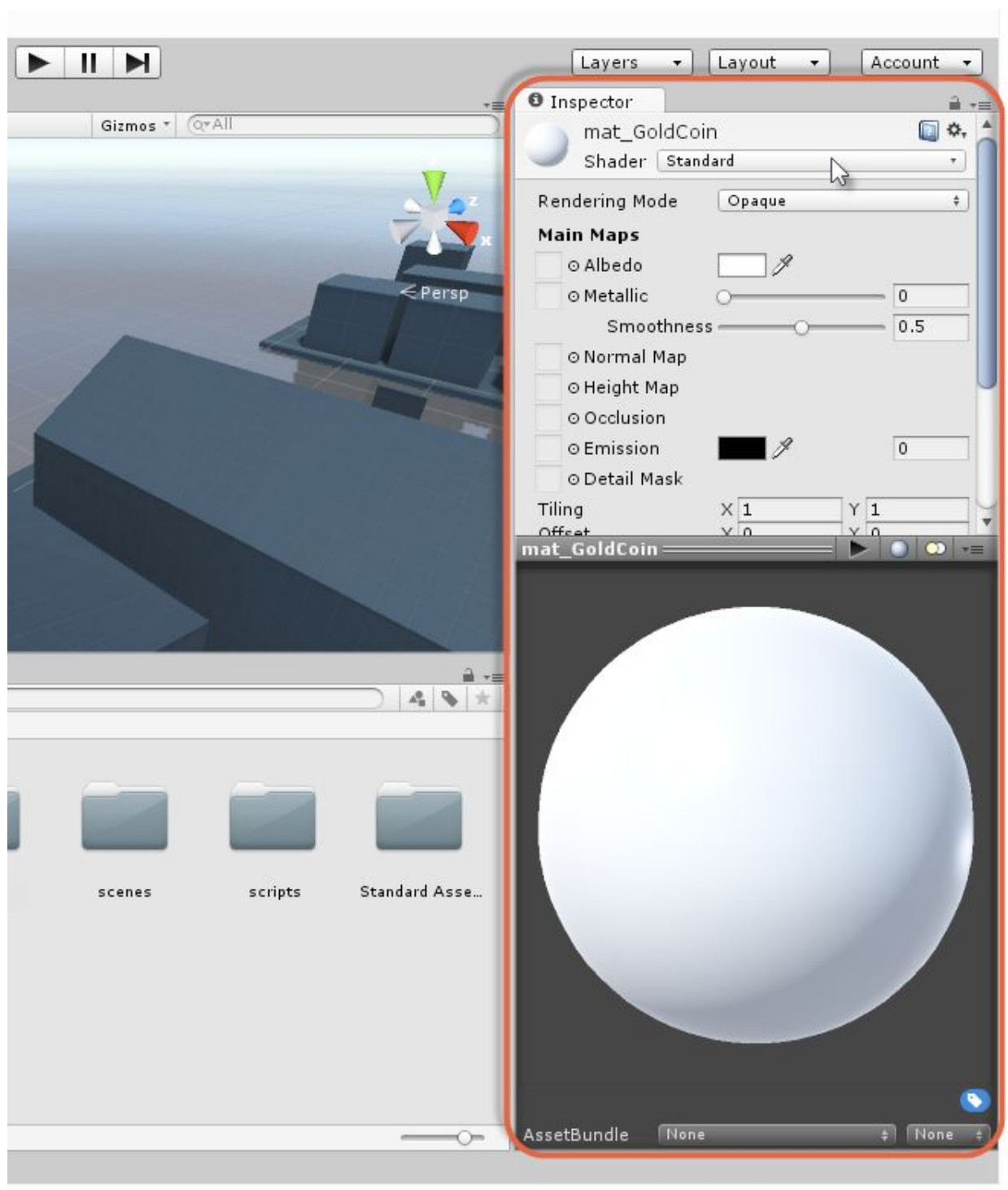


图2.4 从对象检查（Inspector）面板处对材质的属性进行修改

下面为金币创建一个金光闪闪的材质。首先需要指定着色器（Shader）的类型，这是因为这个类型决定了其他可用的参数。着色器（Shader）的类型决定了系统将采用哪种算法来计算对象的阴影。有很多种类型供选择，但是最经常使用的类型就是“Standard”或者“Standard（Specular setup）”。对于游戏中使用的金币，可以将着色器（Shader）的类型设定为“Standard”，如图2.5所示。

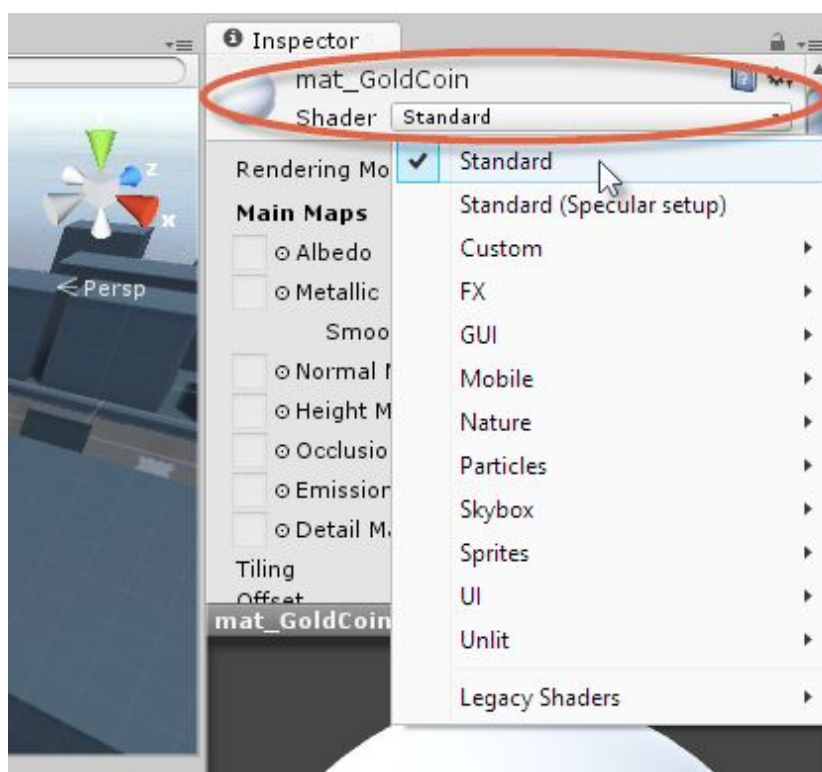


图2.5 设定材质的着色器（Shader）类型

此时，在预览（Preview）面板中显示的是阴暗的灰色，这与所需要的相差太多了。现在需要的是一个金色，首先指定反射（Albedo）的属性。单击反射（Albedo）右侧的色板，然后会弹出一个颜色选择器，在其中选出所需要的金色。这时，反射（Albedo）右侧的色板也会实时根据材质的变化更新，如图2.6所示。

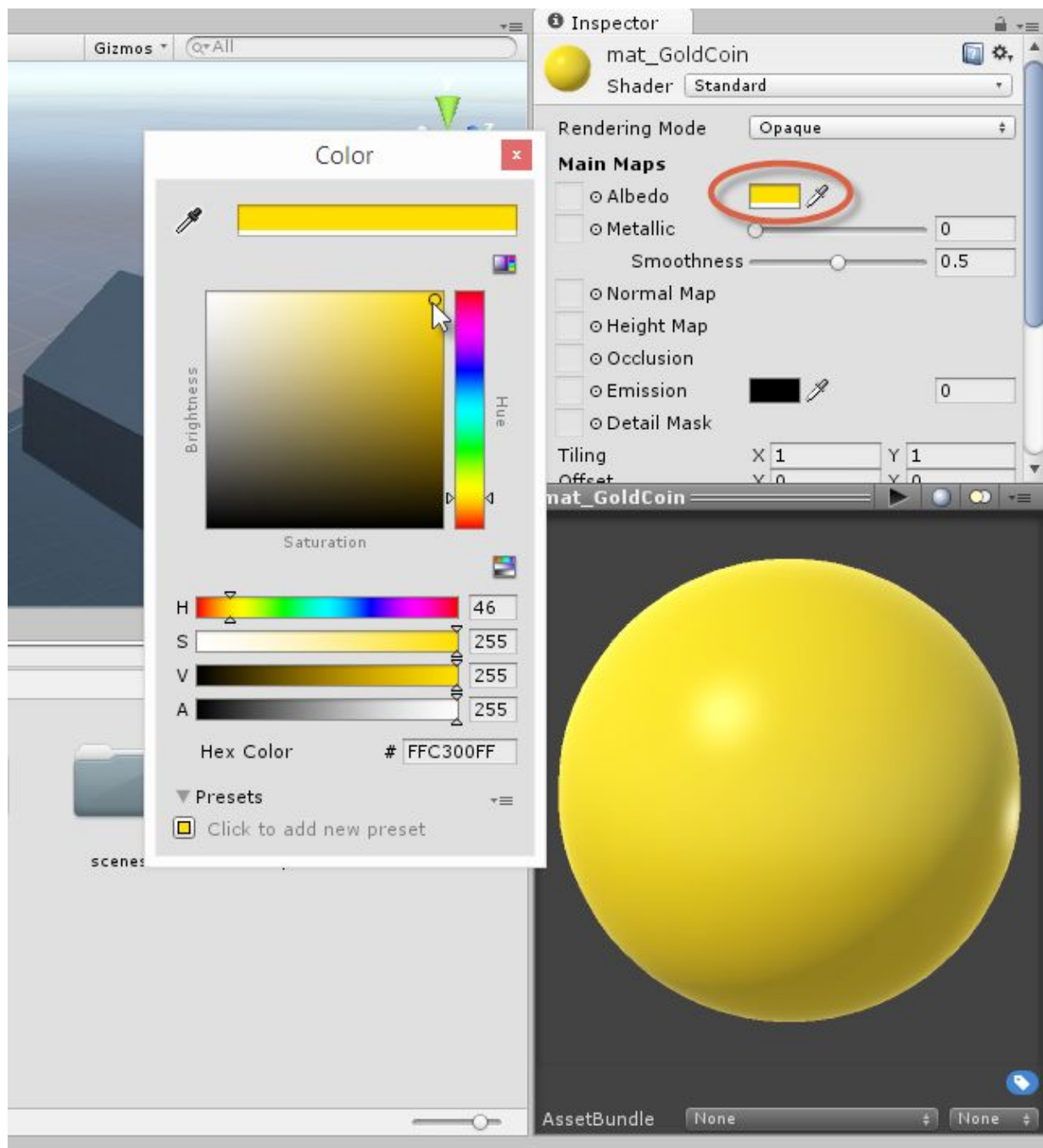


图2.6 将反射（Albedo）的值指定金色

现在，金币的材质比以前好多了，但是需要的游戏对象是金币，也就是说，它们看起来应该是金光闪闪的。为了将这个效果添加到材质库中，可单击并滑动对象检查（Inspector）面板处“Metallic”属性右



侧的滑动条，将值设定为1。这个值设为1表明会将材质设定为一个金属表面，而不是一个像布或者头发之类的漫反射面。同样，此时的预览会如同图2.7所示的一样，及时地根据设定的更改进行更新。

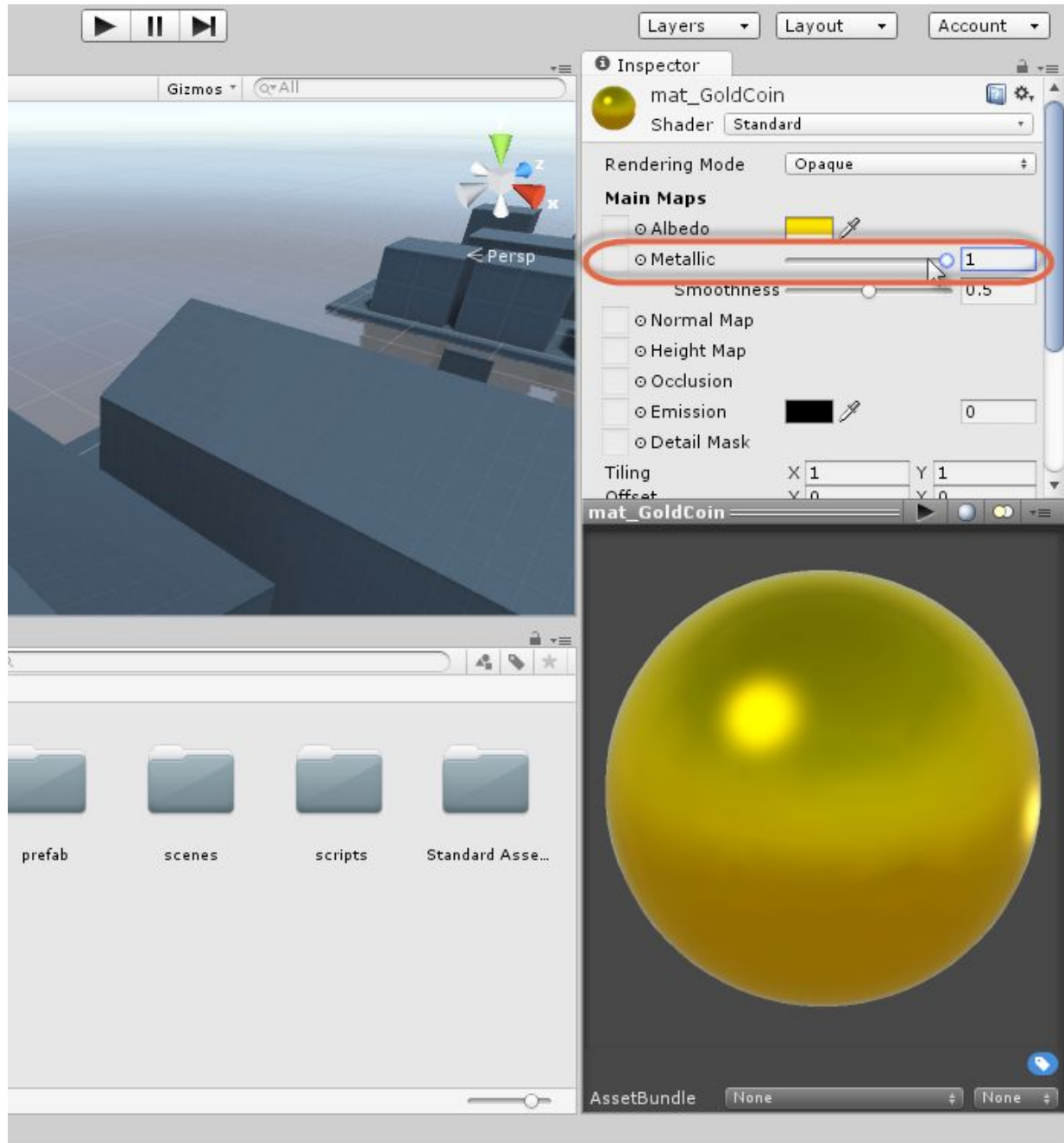


图2.7 创建一个金属材质

现在已经创建好了一个新的金属材质，可以在预览窗口中看到它逼真的效果了。如果需要，还可以在预览窗口中更改材质预览时的形状。默认的情况下，材质是以一个球形展示出来的，不过也可以以其他的基本形状来预览，例如正方体、圆柱体、圆环等。这样有助于在不同的条件下对材质进行预览。可以通过单击预览面板上方的“几何图形”按钮来更改对象的形状（见图2.8）。

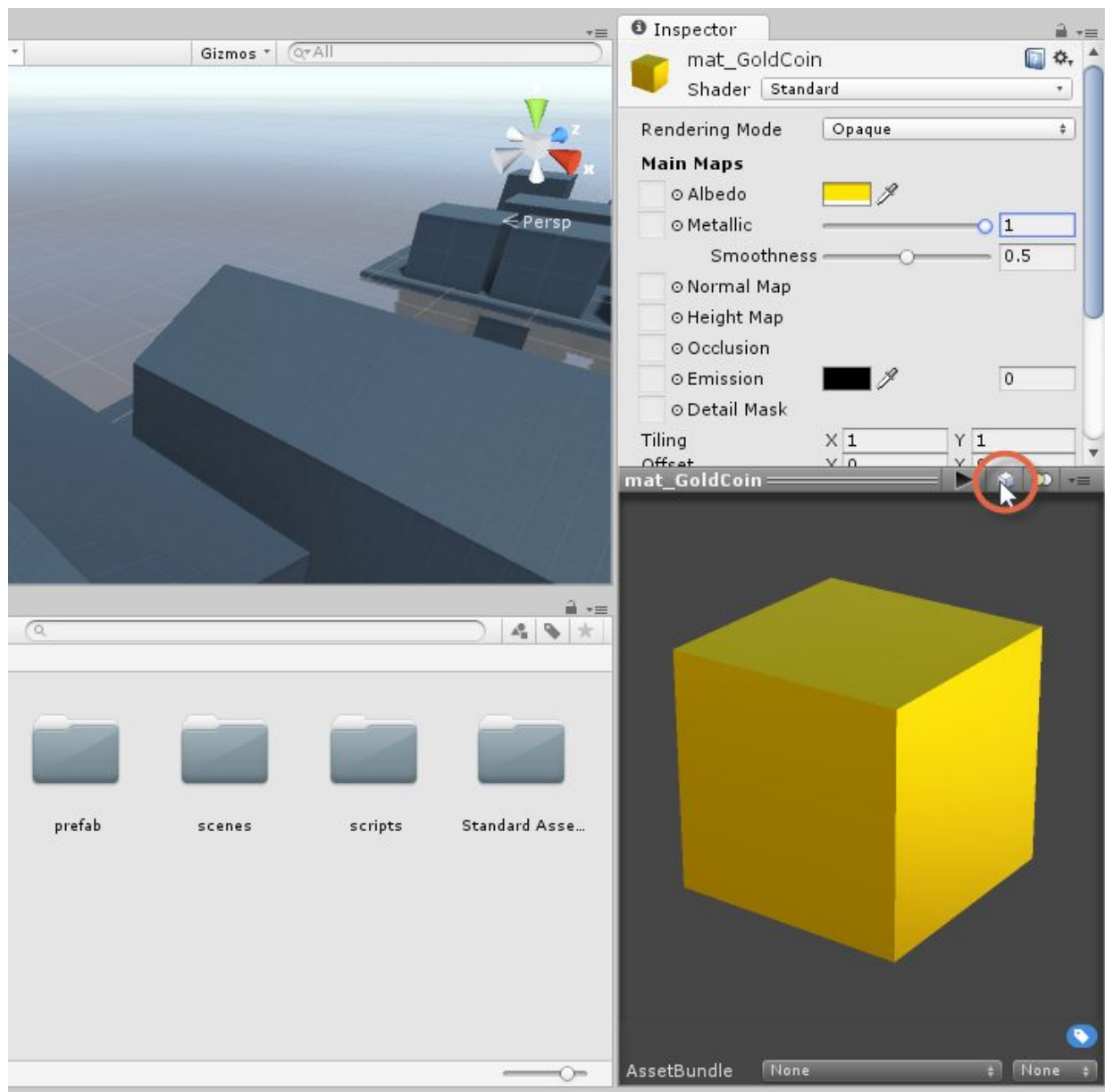


图2.8 对一个对象上的材质进行预览

当为金币准备好了材质之后，就可以通过拖动的方式将这个材质直接应用到场景中的网格上。先将做好的金币材质应用到金币上，从项目（**Project**）面板选中刚做好的材质，然后拖动到场景中的金币对象上。当将材质拖动到金币上释放鼠标时，金币就呈现出金属效果了。这个过程如图2.9所示。

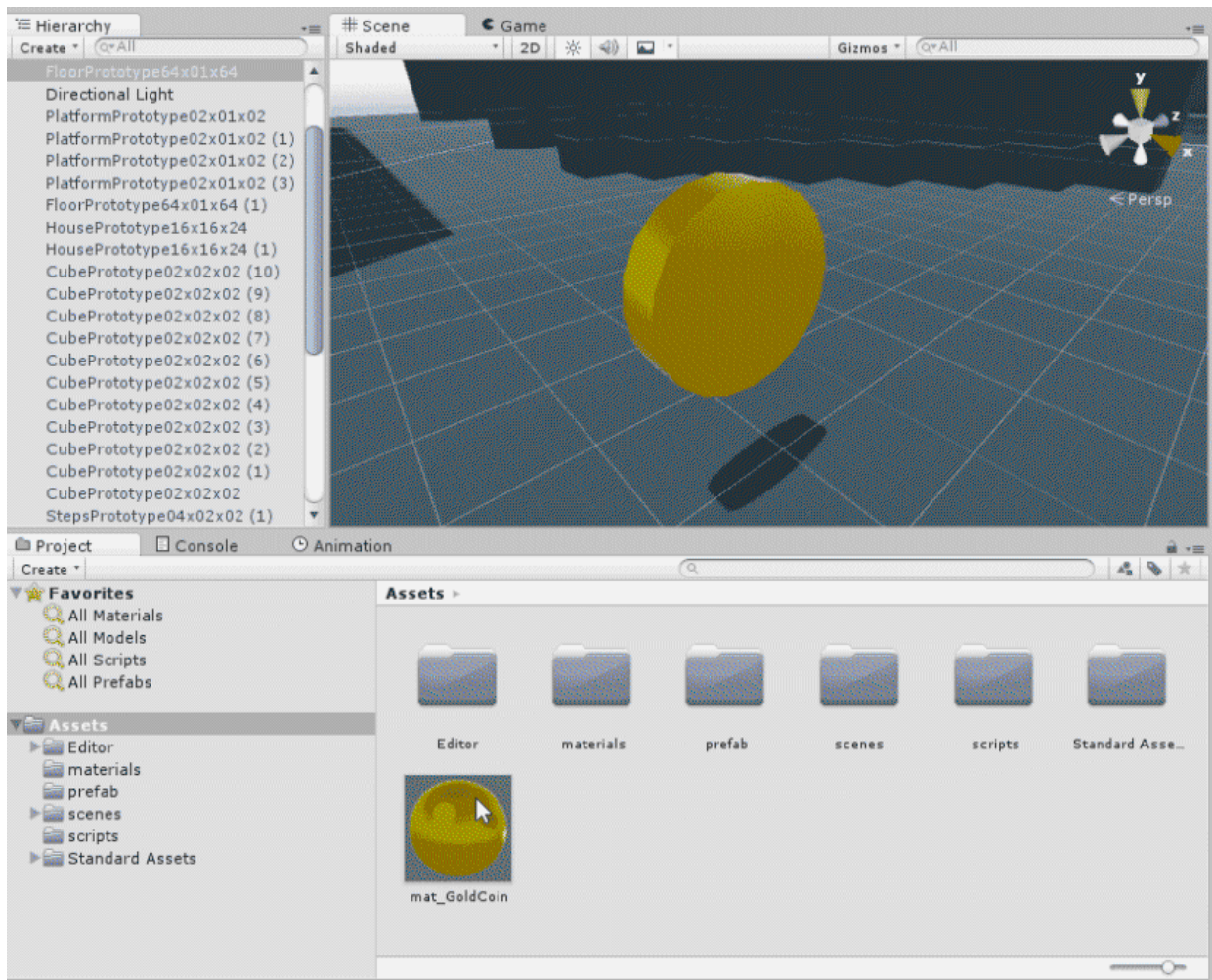


图2.9 为金币设置材质

我们已经成功地为金币添加材质了，也可以通过在场景选中金币对象来查看它所使用的材质了。同样，也可以从检查（Inspector）面板窗口来查看“Mesh Renderer”组件。当使用摄像机时，“Mesh Renderer”组件可以保证一个网格对象是可见的。组件“Mesh Renderer”包含一个材质（Materials）区域，该区域中列出了分配给这个对象的全部材质。在材质（Materials）区域选中材质的名字之后，Unity就会自动地从项目（Project）面板上选中材质，这样就可以快速简单地对材质进行定位，如图2.10所示，组件“Mesh Renderer”列出了分配给一个对象的所有材质。

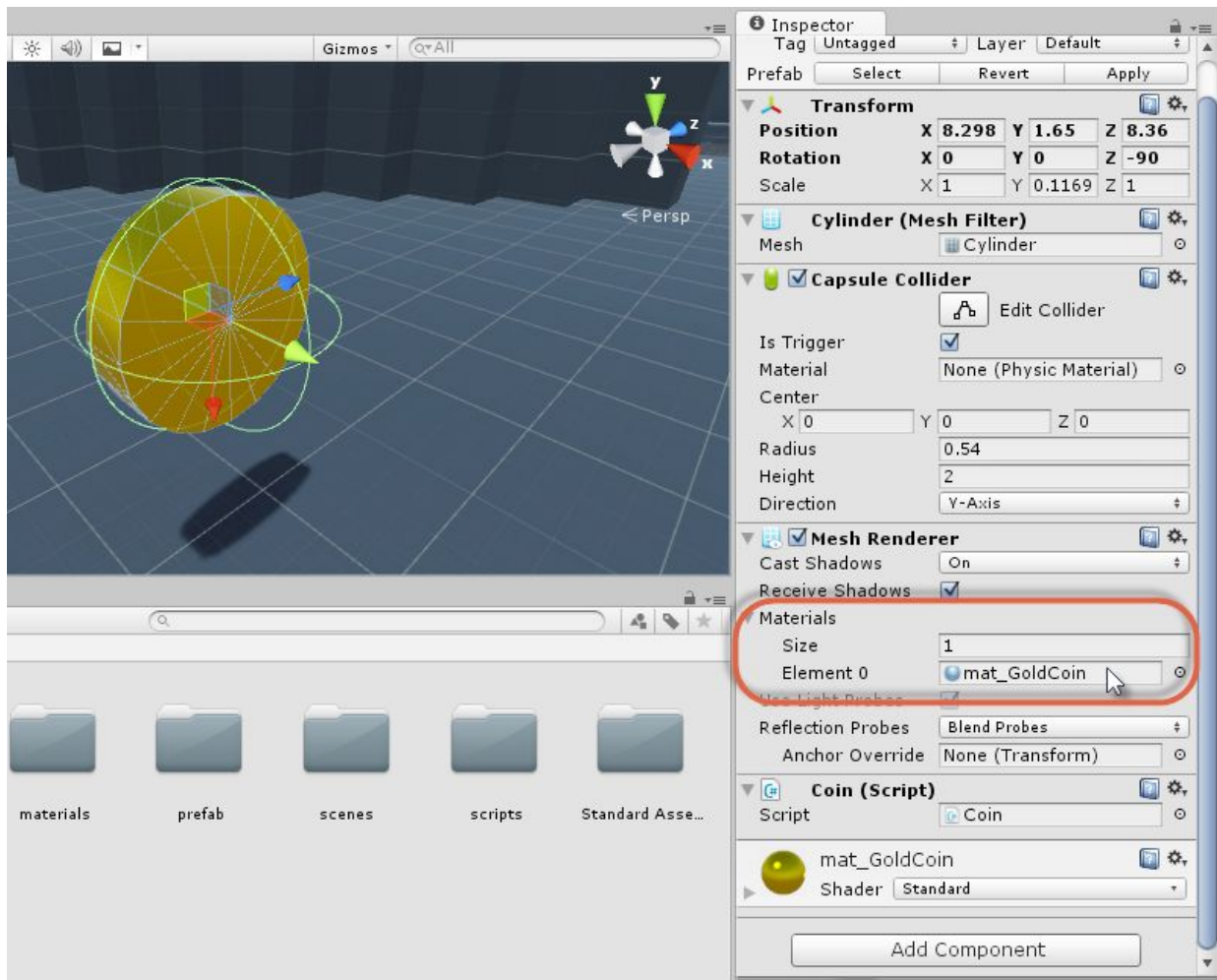




图2.10 组件“Mesh Renderer”列出了分配给一个对象的所有材质



一个网格对象有时会拥有多个材质，对象的每一个面都分配不同的材质。不过为了获得最佳的游戏性能，最好在满足需求的基础上使用尽可能少的材质。如果可能，尽量让多个游戏对象都使用同一个材质，这样做可以很明显地提高游戏的性能。如果想要获得关于游戏渲染性能优化方面的更多内容，可访问<http://docs.unity3d.com/Manual/OptimizingGraphicsPerformance.html>的在线文档。

已经完成可以用来收集的金币的材质了，它看起来很漂亮，不过，并没有彻底地完成金币部分。当玩家接触到金币时，它并不会消失，而且也没有机制来记录玩家到底采集到了多少金币。接下来就需要开始编写程序了。

## 2.2 Unity中的C#脚本

为游戏定义逻辑、规则和行为的时候，往往需要使用到脚本。如果想将那些静态的、无生命的场景和对象转换成为可以进行交互的环境和对象，那么开发人员就需要编写代码。这些代码定义了这些物体在遇到了指定情况之后，应该做出什么样的反应。金币采集游戏也需要编写代码才能实现所有的功能。这个游戏需要实现3个主要的功能：

- 能够感知玩家是否收集到金币；
- 在游戏进行中，能够及时了解到玩家收集的金币数量；



- 能确定游戏时间是否已经结束。

在Unity中并没有包含一个能实现上述功能的模块。所以必须自己来编写一些代码来实现这些功能。Unity中支持两种语言，即UnityScript语言（有时候称之为JavaScript语言）和C#语言。这两种语言功能都很强大，但是本书中主要采用的是C#语言。这是因为从发展的趋势来看，JavaScript的使用率将会逐渐下降。现在开始对这个主要功能进行编程。首先在项目（Project）面板的空白区域单击鼠标右键，然后在弹出的上下文菜单中依次选择“Create | C# Script”，就可以创建一个新的脚本文件。另外，也可以从应用程序菜单处依次选择“Assets | Create | C# Script”来创建一个新的脚本文件，如图2.11所示。

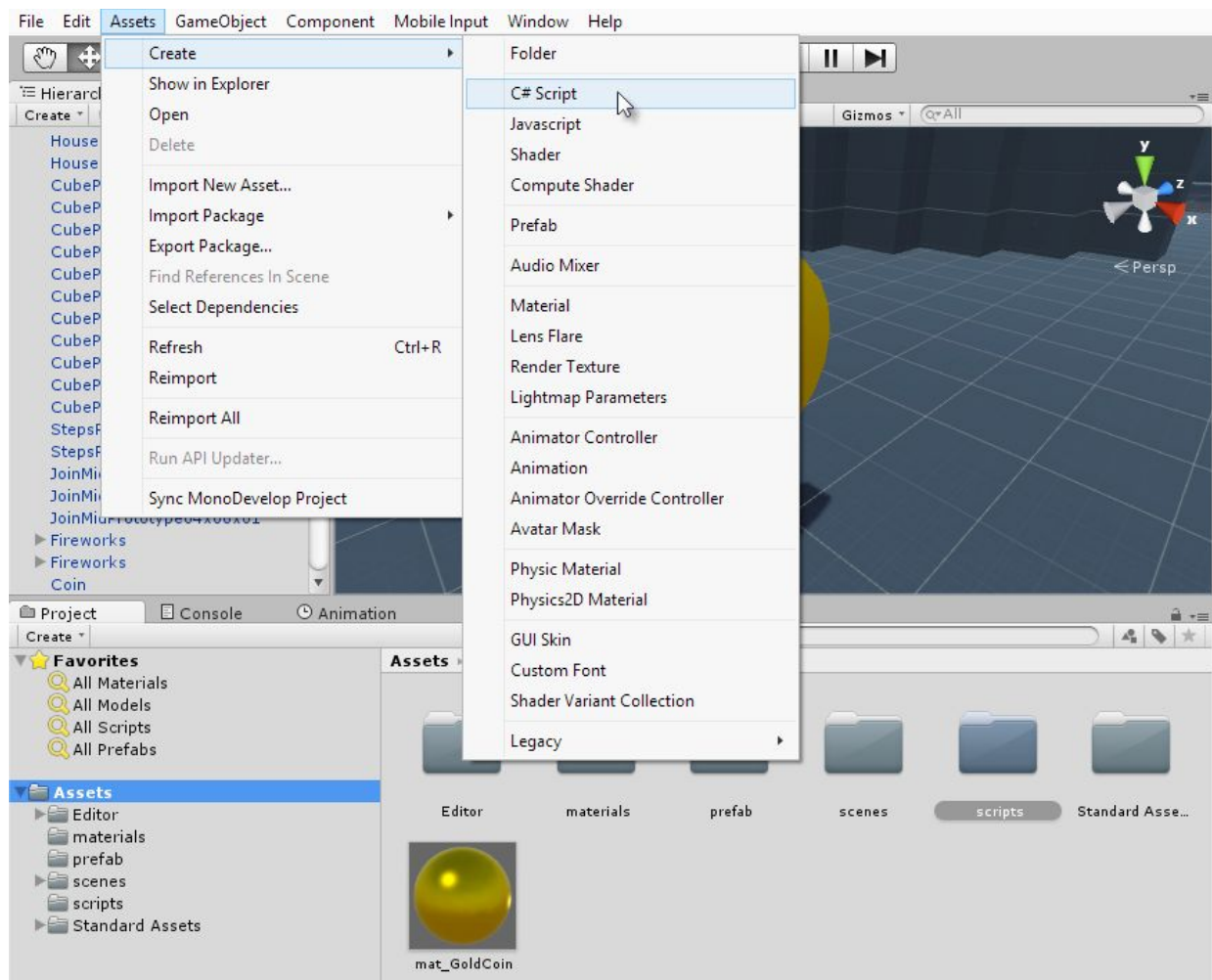


图2.11 创建一个新的C#脚本

当创建了脚本之后，需要为脚本起一个描述性的名字。本书起的名字为“Coin.cs”。在Unity中，每一个脚本文件都对应一个与其同名的类。因此，“Coin.cs”文件对应的就是“Coin”类。这个“Coin”类将封装一个金币的所有行为，并最终会附加到场景中的金币上，如图2.12所示。

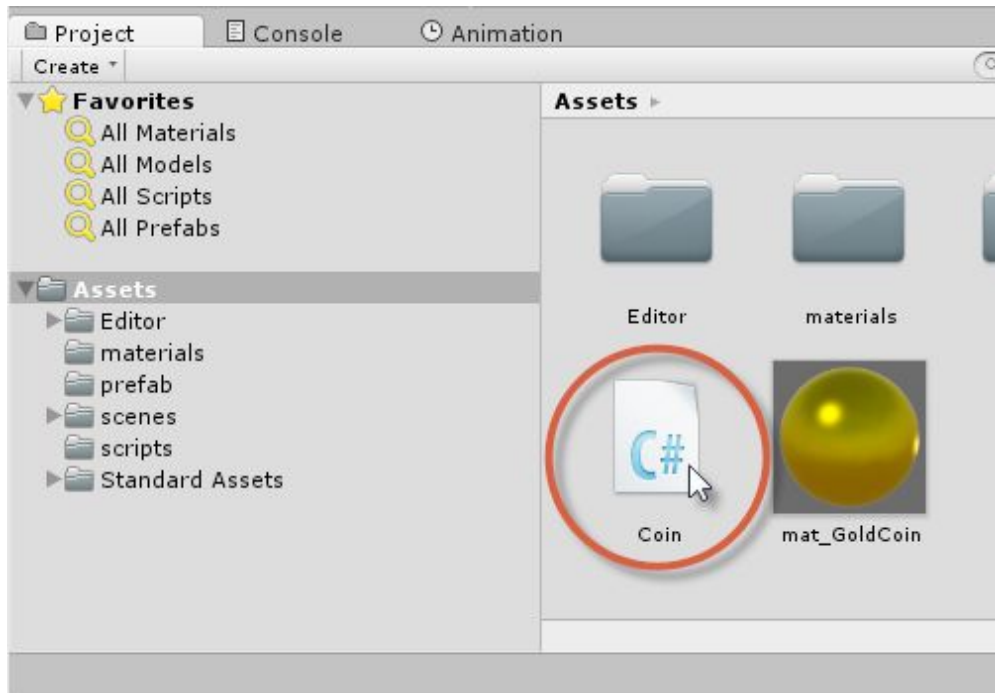


图2.12 为脚本文件命名

在对象检查（Inspector）面板中双击Coin.cs，就可以使用MonoDevelop打开这个文件。MonoDevelop是一款Unity自带的第三方IDE应用，它可以实现对游戏中的代码进行编写和修改。当一个文件在MonoDevelop中打开之后，它的内容就会如代码示例2.1所示的在MonoDevelop中显示出来。

### 代码示例2.1:

```
using UnityEngine;
using System.Collections;

public class Coin : MonoBehaviour
{
    // Use this for initialization
    void Start () {}

    // Update is called once per frame
    void Update () {}
}
```



## 代码示例的下载

可以使用自己的账号从<http://www.packtpub.com>处下载本书的代码示例。无论你在哪里购买的本书，都可以访问<http://www.packtpub.com/support>并进行注册，书中的资源也可以通过电子邮件的形式发送给你。

可以按照如下步骤来下载这些文件：

- 使用电子邮箱地址在页面上注册，如果已经注册过了，那么直接登录即可；
- 找到并使用鼠标单击位于页面顶端的“SUPPORT”；
- 单击“Code Downloads & Errata”；
- 在搜索search框中输入要下载资源的书的名字；
- 选择要下载资源的书；
- 在下拉菜单中选中购买本书的地点；
- 单击“Code Download”。

将这些文件下载了之后，要确定解压缩软件已经更新到了最新的版本：

- WinRAR / 7-Zip for Windows；
- Zipreg / iZip / UnRarX for Mac；
- 7-Zip / PeaZip for Linux。

默认情况下，所有新创建的类都派生自“MonoBehavior”类，这个类中定义了一些对所有组件都通用的函数。“Coin”类具有两个自动生成的函数，也就是Start()和Update()。这些函数都是由Unity自动调用的事

件。当游戏对象（关联了这个游戏脚本）在场景中创建时，就会调用 `Start()` 函数。`Update()` 函数会在每一帧被附加了游戏脚本的对象中调用一次。`Start()` 函数主要用来实现代码的初始化，`Update()` 函数主要用来实现那些随着事件推移的行为，例如运动和变化。现在，将新创建的脚本文件与场景中的金币对象进行管理，可以从项目（**Project**）面板处将“**Coin.cs**”文件拖曳到金币对象上。当完成以后，一个新的金币组件就被添加到了游戏对象上。这意味着这个脚本已经关联到了游戏对象上，如图2.13所示，一个关联了脚本的游戏对象。

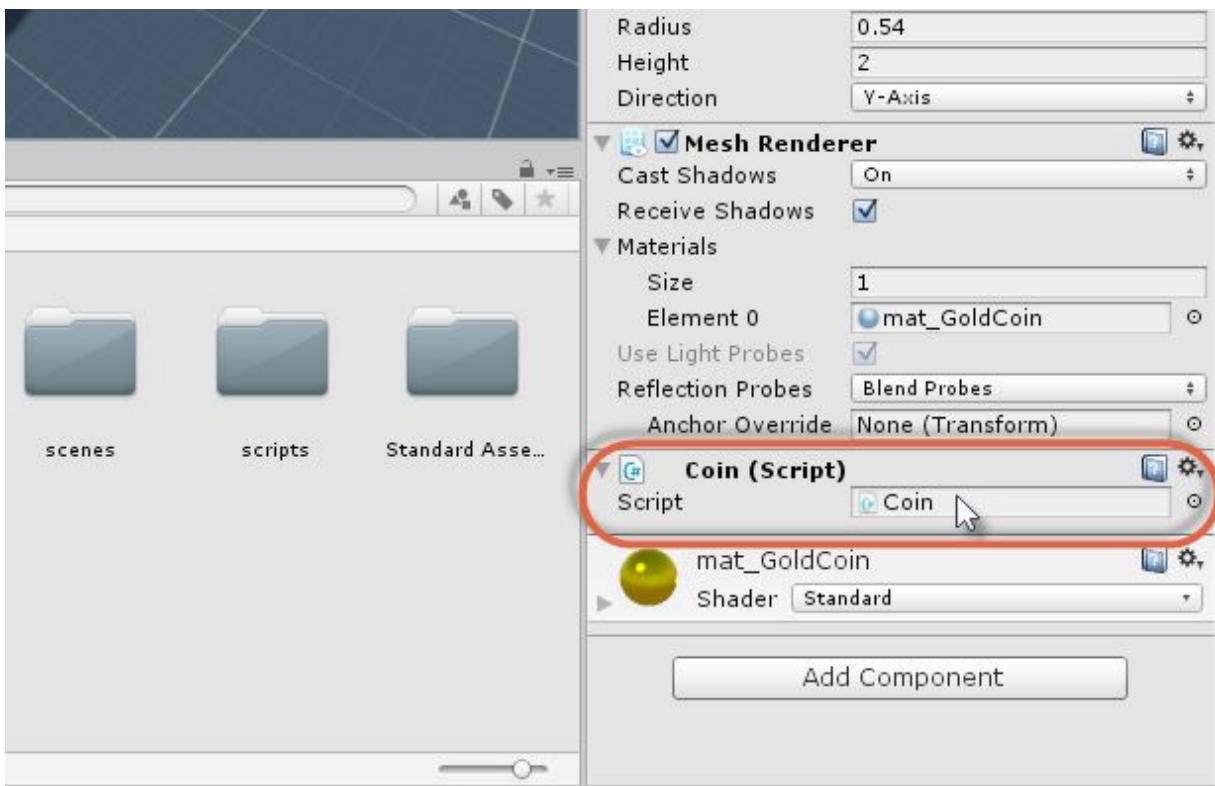


图2.13 一个关联了脚本的游戏对象

当一个脚本与一个游戏对象关联到一起之后，这个脚本就作为这个游戏对象的一个组件而存在。一个脚本文件可以添加到多个游戏对象上，甚至可以被多次添加到同一个游戏对象上。每个组件都代表着



一个单独而且不同的类的实例化。当一个脚本以这种方式添加进来之后，Unity会自动地调用它的函数，例如Start()和Update()。可以在Start()函数中加入一个Debug.Log语句来确认脚本是否能正常工作，这个语句在场景中的游戏对象被创建时在命令行窗口输出一个调试信息。查看如下所示的代码示例2.2。

### 代码示例2.2:

```
using UnityEngine;
using System.Collections;

public class Coin : MonoBehaviour
{
    // start()是初始化函数
    void Start () {
        Debug.Log ("Object Created");
    }

    // Update在每一帧调用一次
    void Update () {

    }
}
```

如果按下工具栏上的“Play”键，或者按下键盘上的“Ctrl + P”组合键，来运行这个向游戏对象上添加了前面的那个脚本的游戏，就会在控制台窗口中看到一条内容为“Object Created”的信息，每当这个类进行实例化的时候，都会输出一次（见图2.14）。

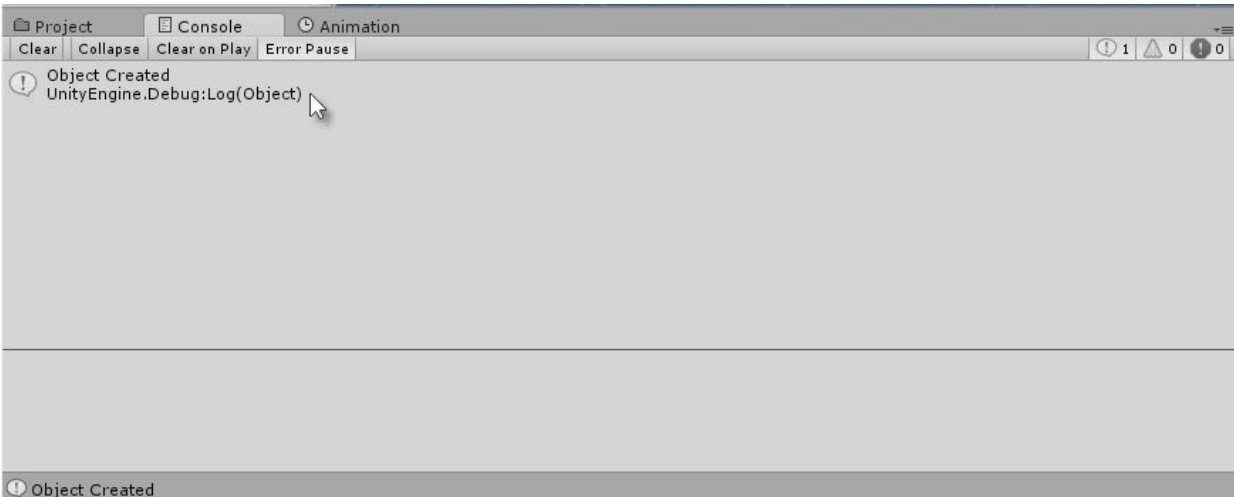


图2.14 在控制台窗口处输出消息

现在已经为Coin类创建了最基本的脚本，并且已经将这个脚本成功附加到了金币对象上。接下来，继续编写一些函数，这些函数将会记录采集过的金币信息。

## 2.3 对金币进行计数

如果整个场景里只有一个金币，那么这个金币采集游戏就有些名不符实了。本游戏的核心设计思路就是在一个关卡中应该包含很多个金币，玩家应该在系统计时结束前收集完这些金币。现在，如果要知道玩家是否将所有的金币都收集齐了，首先得先知道这个场景中总共有多少个金币。毕竟如果不知道金币的总量，也就无法知道是否已经将所有的金币收集全了。所以，第一个任务就是通过Coin类来轻松地知道任意时刻金币的总数。下面给出实现这一功能的Coin类，具体的如代码示例2.3所示。

**代码示例2.3:**

```

//-----
using UnityEngine;
using System.Collections;
//-----
public class Coin : MonoBehaviour
{
    //-----
    //追踪scene中金币数量
    public static int CoinCount = 0;
    //-----
    // start()是初始化函数
    void Start ()
    {
        //创建游戏对象, 增加金币的数量
        ++Coin.CoinCount;
    }
    //-----
    //当游戏对象被销毁时调用OnDestroy函数
    void OnDestroy()
    {
        //减少金币数量
        --Coin.CoinCount;

        //检查剩余金币数量
        if(Coin.CoinCount <= 0)
        {
            //玩家获得胜利
        }
    }
    //-----
}
//-----

```

下面就代码示例2.3进行以下几点总结。

- **Coin**类中有一个静态的成员变量——**CoinCount**。这个变量是静态的，可以被这个类的所有实例所共享。该变量中记录了场景中金币的总数，类中的每个实例都可以访问该变量。
- 每当场景中的一个金币游戏对象被创建时，就会自动调用**Start()**函数。在场景启动时调用**Start**函数就可以设定游戏初始的金币。每

当初始化一个实例，这个函数就会将其中的变量CoinCount的值加1，从而完成对所有金币的计数。

- 每当一个游戏对象被销毁时，就会自动调用OnDestroy()函数。每当一个金币被销毁之后，这个函数就会将其中的变量CoinCount的值减1，从而将金币总量减少1个。

代码示例2.3中包含了一个变量CoinCount，利用这个变量可以知道金币的总量。查询这个值就可以轻松地获得当前金币的总量。现在已经完成整个金币采集功能工作的第一步了。

## 2.4 金币采集

前面已经设计了一个金币计数变量，通过这个变量就可以随时地获知当前场景中金币的数量。不过，现在游戏仍然没有实现金币采集功能。下面就来完善这个功能。当玩家接触到金币后，也就是说，当玩家穿过金币或者碰撞到金币的时候，就可以看作这个金币被成功采集了。

如果想要确定玩家是否与金币发生接触，就需估算出玩家和金币的体积，这样才能检测出两个物体在空间中是否出现重叠。这是Unity通过碰撞体（Collider）来实现的。碰撞体（Collider）是一种附加在网格上的特殊物理对象，当两个游戏对象在空间上有重叠时，碰撞体就会检测到。FPSController对象（第一人称视角控制器）上已经自带了碰撞体对象，它使得控制器更像是一个人的身体。在场景选中FPSController对象（第一人称视角控制器），然后就可以看到围绕在主

摄像头周围的绿色线框，它是胶囊形状的。如图2.15所示，“Character Controller”具有一个和玩家身体形状相仿的碰撞体。

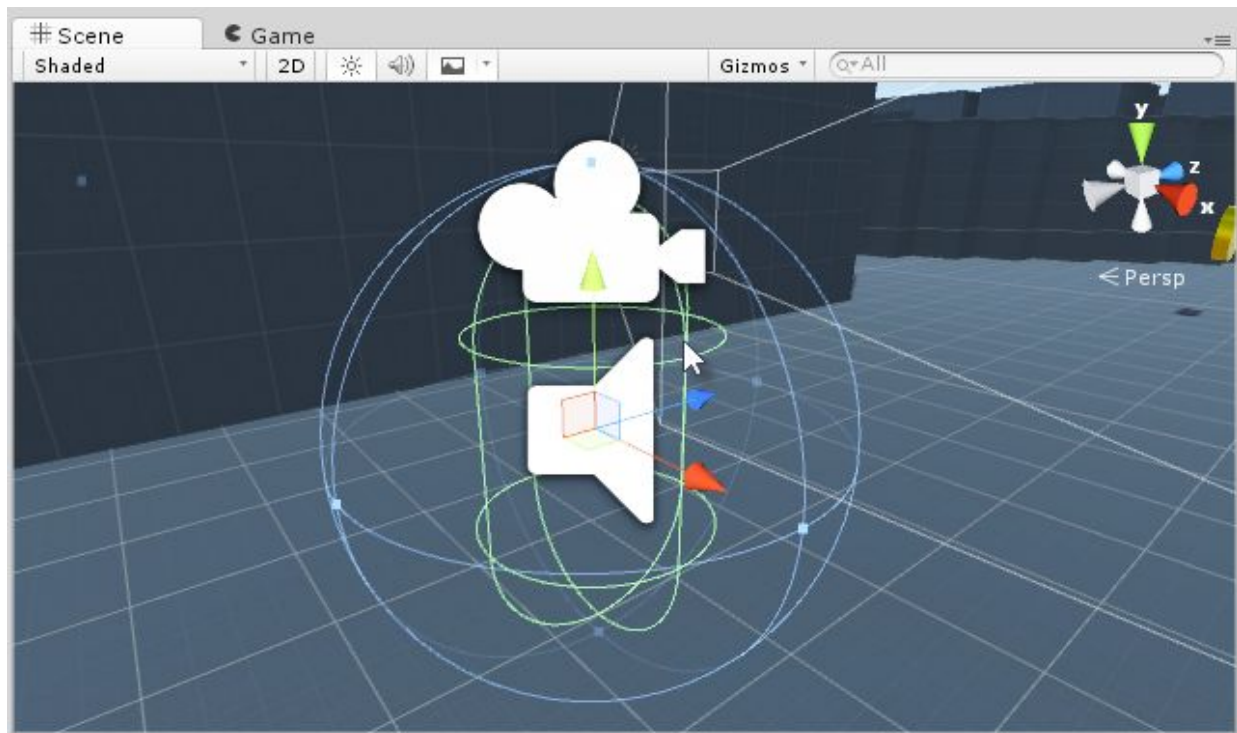


图2.15 “Character Controller”具有一个和玩家身体形状相仿的碰撞体

FPSController 中包含了一个“Character Controller”组件，这个组件包含“Radius”“Height”和“Center”等选项，规定了场景中角色的物理属性。如图2.16所示，FPSController 中已经包含一个“Character Controller”，在此游戏中，这些选项的值都不需要修改，保留默认值即可。



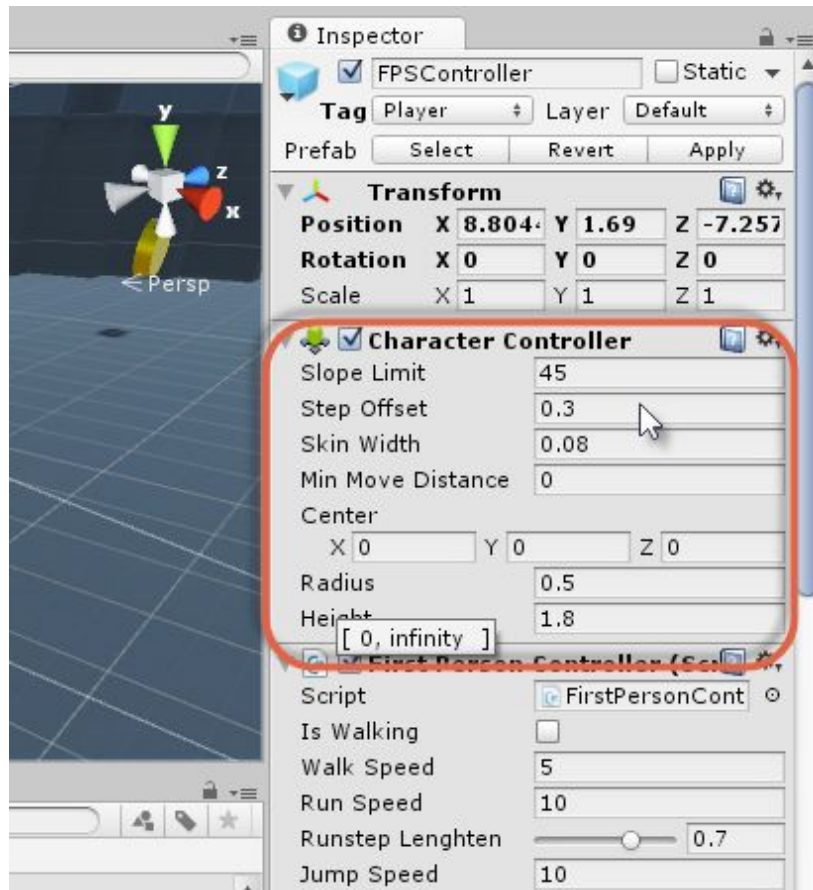


图2.16 FPSController中的“Character Controller”

金币游戏对象中只包含了一个胶囊碰撞体（Capsule Collider）组件，这是当初在创建圆柱体对象时系统自动添加上的，它与场景中的金币的大小差不多，在它的“Character Controller”组件中没有添加任何的属性和动作，这是很正常的，因为与FPSController这种会动的运动物体不同，金币是一个不会动的静态物体。如图2.17所示，包含了胶囊碰撞体（Capsule Collider）组件的圆柱体对象（Cylinder）。

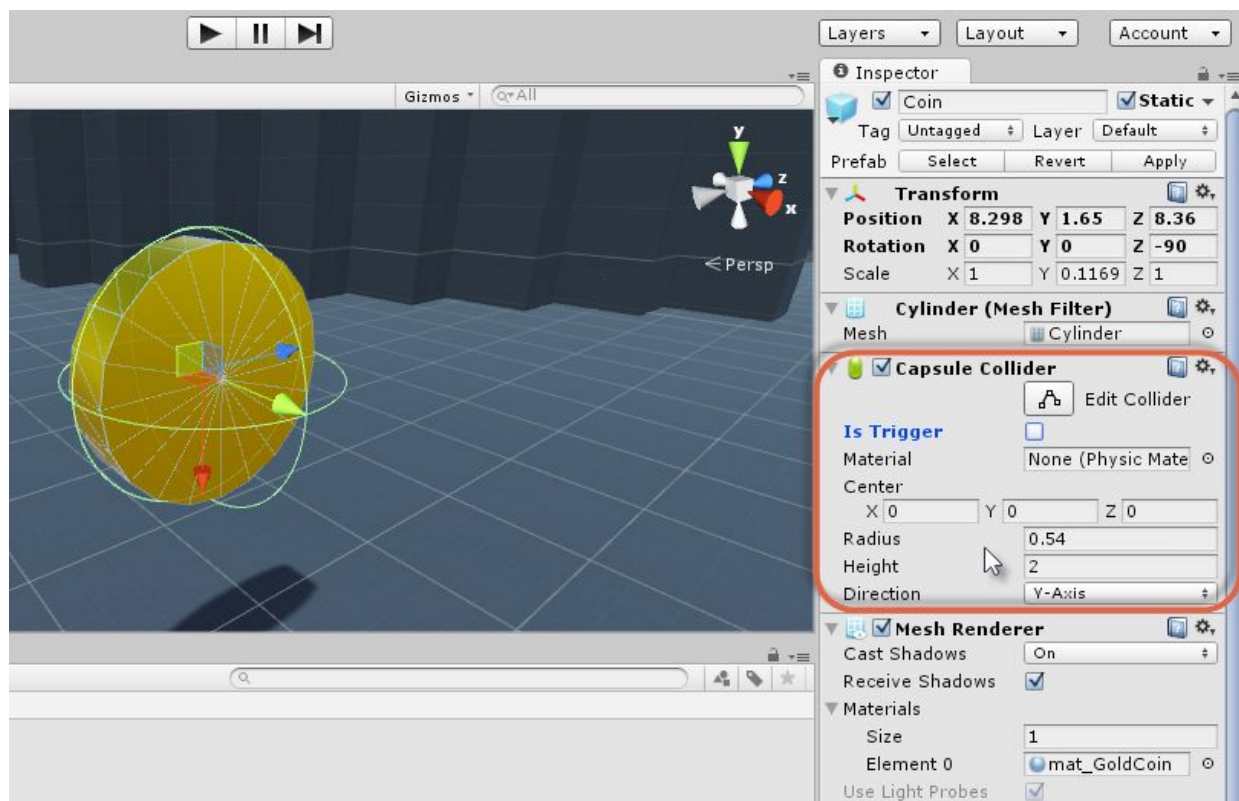


图2.17 包含了胶囊碰撞体（Capsule Collider）组件的圆柱体对象

对于现在这个游戏，仍然为金币游戏对象选择使用胶囊碰撞体（Capsule Collider）组件，如果希望改用一个不同的碰撞体，比如一个盒子或者球形，来代替这个默认组件，就需要首先将金币上任何已有的碰撞体组件移除。具体操作为：单击位于游戏检查（Inspector）面板中组件右上角的齿轮图标，然后从上下文菜单中选中“Remove Component”，如图2.18所示。

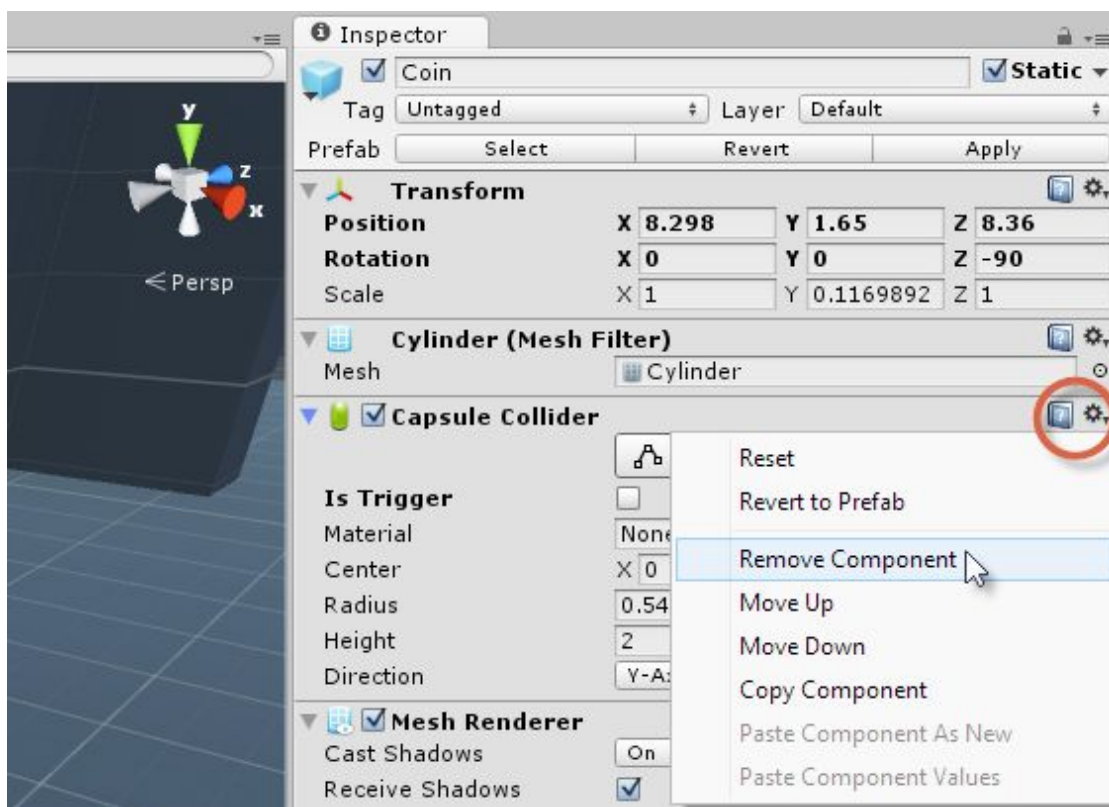


图2.18 从一个游戏对象上将指定组件移除

接下来，选中一个游戏对象，然后通过应用程序菜单依次选中“Component | Physics”，然后在弹出的菜单中选中一个合适形状，如图2.19所示。

图2.19 为选中的游戏对象添加一个组件

不论使用哪种类型的碰撞器，这里都还有一点小问题。那就是当在进行游戏的时候，如果想要穿过金币，就会发现无法成功，反而会被金币挡在那里。这是因为现在的金币是一个固态的物理对象，所以FPSController对象是无法穿金币而过的。但是本游戏可不是这么设计的，金币不应该是一个游戏中的“拦路石”，而是一种可以被采集的对

象，这就是说当玩家从金币中穿过时，就可以采集到金币，同时金币也会在场景中消失。操作方法为：选中金币对象，然后在它的游戏对象检查（Inspector）面板中的胶囊碰撞体（Capsule Collider）组件中选中“Is Trigger”复选框。这个“Is Trigger”选项适用于所有的碰撞体类型，主要用来检测与其他碰撞体是否发生碰撞，并且在发生碰撞时允许它们通过，如图2.20所示。

图2.20 选中“Is Trigger”复选框允许对象穿过其他的碰撞体

此时，在玩游戏时，FPSController就可以轻松地穿过场景中的所有金币。然而，玩家接触到金币之后，金币并不会消失，同样也不会被玩家采集到。要完善这些功能，需要向“Coin.cs”文件中添加更多的代码。首先添加一个OnTriggerEnter()函数，当游戏对象，例如游戏玩家，进入了一个碰撞体时，就会自动地调用。现在，添加一句Debug.Log语句，当玩家进入到一个碰撞体时，就会输出一条调试消息，这只是为了进行测试，代码示例2.4如下所示。

#### 代码示例2.4:

```
//-----  
using UnityEngine;  
using System.Collections;  
//-----  
public class Coin : MonoBehaviour  
{  
    //-----  
    public static int CoinCount = 0;  
    //-----  
    // start () 是初始化函数  
    void Start () {  
        //创建游戏对象，增加金币的数量  
        ++Coin.CoinCount;  
    }  
}
```

```

}
//-----
void OnTriggerEnter(Collider Col)
{
    Debug.Log ("Entered Collider");
}
//-----
//当游戏对象被销毁时调用OnDestroy函数
void OnDestroy()
{
    //减少金币的数量
    --Coin.CoinCount;

    //检查剩余的金币
    if(Coin.CoinCount <= 0)
    {
        //玩家胜利
    }
}
//-----
}
//-----

```



关于OnTriggerEnter()函数的更多详细信息，可以访问Unity的在线文档，访问地址为<http://docs.Unity3d.com/ScriptReference/MonoBehaviour.OnTriggerEnter.html>

首先单击工具栏上的“Play”按键来测试样例2.4中的代码。当穿过一个金币时，就会触发这个函数并展示一个信息。不过，这里还是存在一个问题，到底是什么触发了这个函数呢？很明显一定有什么东西穿过了金币，但是到底是什么呢？是玩家，敌人又或者是一个掉落的砖头，还是其他的什么东西？为了区分开这些内容，需要使用“Tag”。“Tag”可以为场景中的指定物体打上特殊的标记或者标签，这样就可以在代码中轻松将玩家和其他对象区分开。要知道在本游戏设



计中，只有玩家才可以收集金币。所以，需要为玩家对象添加一个名为“Player”的“Tag”。要做到这一点，需要首先在场景中选中这个对象，然后在对象检查（Inspector）面板中选中Tag下拉列表框，在Tag的下拉列表中选中“Player”。这样就将FPSController标记为“Player”对象，如图2.21所示。

图2.21 为FPSController添加一个“Player”标签

FPSController已经被标记为了Player。接下来就按照代码示例2.5所示重新定义Coin.cs类。现在这个类已经可以完成金币的采集工作，当玩家接触到金币后，金币就会消失，同时将金币的数量减1。

### 代码示例2.5:

```
//-----  
using UnityEngine;  
using System.Collections;  
//-----  
public class Coin : MonoBehaviour  
{  
    //-----  
    public static int CoinCount = 0;  
    //-----  
    // start()是初始化函数  
    void Start () {  
        //创建游戏对象，增加金币的数量  
        ++Coin.CoinCount;  
    }  
    //-----  
    void OnTriggerEnter(Collider Col)  
    {  
        //如果玩家采集到了金币，就销毁这个对象  
        if(Col.CompareTag("Player"))  
            Destroy(gameObject);  
    }  
    //-----  
    //当游戏对象被销毁时调用OnDestroy函数
```

```
void OnDestroy()  
{  
    //减少金币的数量  
    --Coin.CoinCount;  
  
    //检查剩余的金币  
    if(Coin.CoinCount <= 0)  
    {  
        //玩家胜利  
    }  
}  
//-----  
}
```

下面对代码示例2.5进行如下总结。

- 每当FPSController接触到了金币碰撞体的时候，Unity就会自动地调用一次OnTriggerEnter()函数。
- 当调用OnTriggerEnter()函数时，参数Col中存储在这次接触中进入碰撞体的物体信息。
- 函数CompareTag()的作用是用来判断到底与这个碰撞体发生碰撞的是不是Player。
- 使用Destroy函数来销毁一个金币对象。当这个函数被调用时，还会触发OnDestroy()函数，这个事件会使得金币总数减1。

至此，已经创建好了第一个可以工作的金币了。现在游戏中的玩家就可以走近并采集这个金币，然后这个金币就会被从场景中移除。不过，这个场景中不应该只包含一个金币，而是很多个金币。可以通过对这个已经做好的金币进行复制来解决这个问题，当然要将复制好的金币放置到其他不同的地方。其实还有更好的解决办法，接下来将进行介绍。

## 2.5 金币与预设体

现在已经搞定了金币的基本功能，但是整个场景中还需要更多的金币。上一节提出了一个简单的解决方案，但是问题是如果这么做了，当需要对金币的属性进行改变时，那么将不得不逐个对所有金币进行修改。为了避免这种繁琐的重复性工作，可以使用Unity中的预设体（Prefabs）。预设体功能可以将一个场景中的对象转换为一个项目（Project）面板上的资源。预设体通常是场景实例化中最经常需要的资源。这样做的优势就在于，当对资源进行更改的时候，可以将修改自动地应用在一个场景甚至多个场景中所有的实例上。

这样将会使得在使用自定义的资源进行工作时变得更简单，所以现在先来将金币设置为预设体。首先选中场景中的金币对象，然后将它拖曳到项目（Project）面板中。当完成了这个步骤之后，一个新的预设体就做好了。场景中的对象就会升级成为一个预设体的实例。但是这也意味着，一旦这个资源从项目（Project）面板上删除，所有该资源对应的实例也就会变得无效，如图2.22所示。

图2.22 创建一个金币的预设体

现在预设体已经成功创建了，可以从项目（Project）面板中向场景中拖曳预设体来创建更多的金币实例了。每一个金币的实例都和初始的预设体资源相关联，当对预设体资源做出改动时，这些改变会立刻在所有的实例上生效。按照这个思路，向游戏中添加足够的金币对象。图2.23所示为设计的金币分布。

图2.23 向关卡中添加金币的预设体

这里又产生了一个新的问题，即如何才能将一个预设体的实例变回一个独立的游戏对象，而不再关联到预设体资源。这是非常有用的，例如有些时候，可能需要利用预设体来实例出一些对象，但是又不希望这些对象和预设体关联在一起。这一点也很容易实现，首先在场景中选中一个预设体的实例，然后在应用程序菜单中选中“GameObject | Break Prefab Instance”，如图2.24所示。

图2.24 “Break Prefab Instance”功能



当向场景中添加了一个预设体的实例并对其进行了改动之后，如果想将这个改动应用到预设体资源中，就可以选中对象，然后在应用程序菜单中选中“Game Object | Apply Changes to Prefab”。

## 2.6 定时器和倒计时

现在已经拥有一个具备场景和金币的关卡了。凭借新添加进来的Coin.cs脚本，现在的金币也可以被正常计数和采集了。不过对于玩家来说，现在的关卡中仍然没有挑战性，因为他们既赢不了，也输不了。在这个游戏中，玩家没有需要达成的目标。因此时间限制也就成

为了游戏中的一个重要条件：它定义了玩家胜利和失败的关键。也就是说，在游戏时间结束前，玩家采集到了所有的金币，就可以看作玩家胜利；玩家没有采集到所有的金币，则玩家失败。所以需要为这个关卡创建一个倒计时的计数器。首先选择菜单栏上的“GameObject | Create Empty”创建一个新的空游戏对象，并为其起名为“Level Timer”，如图2.25所示。

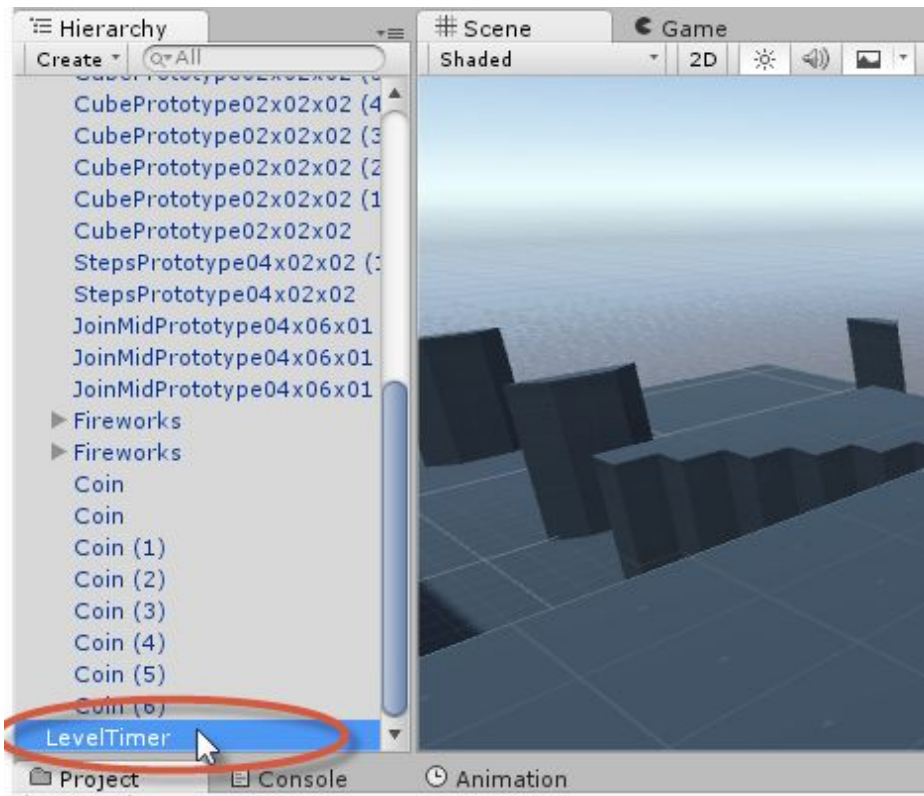


图2.25 为计时（Timer）对象重命名



记住，游戏开始时，玩家是看不到任何空游戏对象的，因为这些对象身上没有任何的网格渲染组件。但是这些对象在用来创建一些不直接对应



到物理的和可见实体上的功能和行为时，就显得特别有用了，例如定时器、管理器和游戏逻辑控制器之类。

接下来，创建一个名为“**Timer.cs**”的新脚本文件，然后将这个脚本添加到场景中的**LevelTimer** 对象上。完成这个操作以后，在这个场景中，定时器就开始起作用了。不过需要确定的是，这个定时脚本仅仅被添加到了场景中的一个物体上，而不是多个物体上。否则，在同一场景中可能就会出现多个相互矛盾的定时器。可以使用层次

（**Hierarchy**）面板来查找整个场景中特定类型的组件。首先在层次

（**Hierarchy**）查找框中输入“**t:Timer**”，然后按下键盘上的回车键开始查找。这样将会查找到场景中所有带有定时器类型组件的对象，并将在层次（**Hierarchy**）面板中显示查找的结果。另外，在这个层次

（**Hierarchy**）面板中只会显示匹配的结果。在查找时输入的“**t**”表示这次查找的条件是类型，如图2.26所示。

图2.26 查找具有特定类型组件的对象

如果在查找过程中停止这次查找，并返回层次（**Hierarchy**）面板的初始状态，则可以通过单击查找对话框右侧的“×”号图标按钮来实现。不过这个按钮很难发现，如图2.27所示。

图2.27 取消查找

如果希望Timer脚本生效，那么必须在里面编写相应的代码。下面的代码示例2.6给出了Timer.cs中的完整代码。如果读者之前从未在Unity中进行过系统的编码，那么这段代码将会是非常有学习价值的。这段代码给出了很多十分重要的特点，代码中的注释给出了详细的说明。

### 代码示例2.6:

```
//-----  
using UnityEngine;  
using System.Collections;  
//-----  
public class Timer : MonoBehaviour  
{  
    //-----  
    //完成关卡的时间（单位为秒）  
    public float MaxTime = 60f;  
    //-----  
    //倒计时  
    [SerializeField]  
    private float Countdown = 0;  
    //-----  
    // start()是初始化函数  
    void Start ()  
    {  
        Countdown = MaxTime;  
    }  
    //-----  
    // 每一帧都会调用update  
    void Update ()  
    {  
        //时间减少  
        Countdown -= Time.deltaTime;  
  
        //当时间耗尽之后重启关卡t  
        if(Countdown <= 0)  
        {  
            //重置金币的数量  
            Coin.CoinCount=0;  
            Application.LoadLevel(Application.loadedLevel);  
        }  
    }  
    //-----  
}
```

```
}  
//-----
```

现在对上面的代码进行如下总结。

- 在Unity中，将类变量声明为公共变量（例如public float MaxTime）之后，会作为一个可以编辑的字段显示在Object Inspector处。虽然这并不支持所有的数据类型，但是却是相当有用的一个功能。这意味着开发人员可以直接从Inspector处监视公共变量的值，或者设置公共变量的值，而不是每次都要对代码进行修改和编译。默认情况下，在Inspector处私有变量是看不到的。不过，如果需要，可以使用SerializeField属性强制使它们可见。如果私有变量前面加上了这个属性，例如CountDown变量，就会在不改变变量的作用范围前提下，像一个公共变量一样显示在对象Inspector中。
- 所有由MonoBehaviour派生出来的类都支持Update()函数，Scene中所有Active的游戏对象在每一帧中都会自动调用这个函数。简而言之，Update()函数会在每秒执行很多次。游戏的FPS是一个每秒钟执行次数的指标。而实际这个数字可能在每秒钟都会发生变化，Update函数在随着时间的变化去改变对象时相当有用。例如在使用CountDown类时，追踪时间的变化是相当有用的。如果了解更多的关于Update函数功能，可以访问

[http://docs.Unity3d.com/ScriptReference/MonoBehaviour.](http://docs.Unity3d.com/ScriptReference/MonoBehaviour.Update.html)

[Update.html](http://docs.Unity3d.com/ScriptReference/MonoBehaviour.Update.html)这个网址上的Unity在线文档。



除了在每一帧都会被调用的Update()函数，Unity中还支持两个相关的函数，即FixedUpdate()和LateUpdate()。FixedUpdate()主要用来实现物理编码，稍后将看到，这个函数在每一帧都会被调用相同的次数。LateUpdate()函数会被每个活动（Active）的对象在每一帧调用一次，但是LateUpdate()函数的调用总是发生在Update()事件之后。因此，它总是发生在Update()周期之后，使它成为一个后期Update。在本书的后面将介绍这个后期Update()存在的特殊用途。如果了解更多的关于FixedUpdate()函数功能，可以访问<http://docs.Unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html>。在线文档。同样，如果了解更多的关于LateUpdate函数功能，可以访问<http://docs.Unity3d.com/ScriptReference/MonoBehaviour.LateUpdate.html>。在线文档。

- 当进行编程时，Time.deltaTime是一个被Unity不断更新的静态变量。它用来描述自从上一帧结束之后到现在的时间量（以秒为单位）。例如游戏的帧速率是2FPS（这是一个非常低的值），deltaTime的值就是0.5。这是因为在每一秒钟将会有两帧，因此每一帧就是半秒。deltaTime是非常有用的，因为随着时间的推移，这个值会体现出自从游戏开始到现在经过了多少时间。deltaTime作为一个浮点型变量，主要用在Update函数中，用来从总时间中减去已经经过的时间。如果了解更多关于deltaTime的信息，可以访问<http://docs.Unity3d.com/ScriptReference/Time-deltaTime.html>上的在线文档。
- 静态函数Application.LoadLevel可以在代码的任何地方被调用，用来改变在运行时的场景。这个函数在将游戏从一个关卡切换到另一个关卡时是相当有效的，它将会使Unity终止当前活动的场景，销毁其中的所有内容，然后加载一个新的场景。另外，这个函数也可以通过再次加载当前活动的关卡来重新启动这个场景。Application.LoadLevel十分适合于那些明确定义了关卡的游戏，这

些游戏的开始与结局都有明确的分界线。不过它并不适合于那种大型的开放性的游戏，这种游戏的世界看起来都是无限伸展的，看起来没有任何的断裂。关于Application.LoadLevel的更多信息可以在

<http://docs.unity3d.com/ScriptReference/Application.LoadLevel.html>上的Unity在线文档中找到。

完成Timer脚本的创建工作之后，在场景中选中LevelTimer对象。从对象的检查（Inspector）面板中就可以看到玩家在当前关卡完成任务所允许的最大时间限制（以秒为单位），如图2.28所示。这里将时间设置为60秒。这意味着玩家需要在关卡开始之后的60秒之内完成所有金币的采集工作。如果定时器中的时间耗尽，那么这个关卡将会重启。

图2.28 设定当前关卡的时间限制

现在有了一个具备倒计时功能的完整关卡，可以完成金币的采集，而且定时器也可以耗尽时间。总的来说，现在的游戏已经初具规模了，不过这里还有一个更深入的问题，后面内容将进行讲解。

## 2.7 庆典和焰火

至此，金币采集游戏几乎要完成了。金币可以被采集了，计时器也开始工作了，但是胜利的条件却没有得到妥善的处理。当玩家在计时器中的时间耗尽之前采集到所有的金币，游戏中并没有任何胜利的提示。相反倒计时仍然会继续，就好像胜利的条件完全没有达到一

样。下面来解决这个问题，首先，当游戏者在游戏中获得胜利之后，应该删除计时器对象，以防止倒计时继续进行，并给出一些视觉上的效果来提示玩家该关卡已经完成。在这种情况下，可以添加一些焰火（Fireworks）。从Unity 5的例子系统包（Particle System Packages）即可添加焰火，操作方法为：依次选中“Standard Assets | Particle Systems | Prefabs folder”，然后将焰火粒子系统（Fireworks Particle System）拖曳到场景中，可以继续添加2到3个焰火粒子系统，如图2.29所示。

图2.29 添加两个焰火预设体

默认情况下，所有的焰火粒子系统都会在关卡一开始时执行。可以通过在工具栏上单击“Play”按钮来测试这个游戏。不过这并不是所希望的效果，我们所设计的效果应该是当游戏者达成胜利条件以后，才执行这个焰火的特效。在场景中选中焰火粒子系统对象，然后在对象的 Inspector 面板上取消组件“Particle System”上单选框“Play On Awake”的选中状态，取消“Play On Awake”的功能，如图2.30所示。

图2.30 取消“Play On Awake”功能

取消“Play On Awake”之后，粒子系统就不会在关卡开始的时候自动执行了。这是正确的，但是这些粒子系统在整个游戏中都不会执行了，应该在合适的时候启动这些粒子系统，通过代码的编写就可以实现这一点。在进行编码之前，首先将全部的焰火对象都赋予一个合适的标签。这样做的原因是，在代码中需要在场景中查找全部的焰火对



象，然后触发对应的事件。为了将焰火对象和其他对象区分开，可以使用Tag属性。所以，首先创建一个新的焰火Tag，然后将它们分配给场景中的焰火对象。在这一章开始的时候就为了检测是否与硬币发生碰撞为游戏中的角色添加过Tag属性，如图2.31所示。

图2.31 为焰火对象添加Tag

现在所有的焰火对象都已经被添加了Tag了，接下来可以重新定义Coin.cs脚本类来处理场景中的胜利条件，如下面的代码示例2.7所示。

### 代码示例2.7:

```
//-----
using UnityEngine;
using System.Collections;
//-----
public class Coin : MonoBehaviour
{
    //-----
    public static int CoinCount = 0;
    //-----
    // 初始化函数
    void Awake ()
    {
        //创建对象并增加金币的数量
        ++Coin.CoinCount;
    }
    //-----
    void OnTriggerEnter(Collider Col)
    {
        //当玩家采集到金币之后要销毁这个金币对象
        if(Col.CompareTag("Player"))
            Destroy(gameObject);
    }
    //-----
    void OnDestroy()
    {
        --Coin.CoinCount;
    }
}
```

```

//检查剩余金币的数量
if(Coin.CoinCount <= 0)
{
    //玩家收集完全部金币，游戏胜利
    //销毁计时器对象，启动庆祝焰火
    GameObject Timer = GameObject.Find("LevelTimer");
    Destroy(Timer);

    GameObject[] FireworkSystems =
        GameObject.FindGameObjectsWithTag("Fireworks");
    foreach(GameObject GO in FireworkSystems)
        GO.GetComponent().Play();
}
}
//-----
}
//-----

```

下面对代码示例2.7进行总结。

- **OnDestroy()**函数是十分重要的，当一个金币被采集之后就会调用这个函数，它有一个用来判断是否全部金币都被收集（获胜条件）的if语句。
- 当获胜条件达成之后，系统就会调用函数**GameObject.Find**来寻找所有名为**LevelTimer**的活动对象。如果找到了符合条件的对象，这个对象就会被删除。这通常发生在玩家已经在当前关卡取得胜利的情况下，需要删除计时器以防止它继续倒计时。如果场景中包含了多个符合条件的对象，那么该函数只会返回第一个对象。所以在同一个场景中只能有一个计时器。



需要注意的是，要尽可能避免使用**GameObject.Find()**这个函数，它的执行效率很低。相对而言，**FindGameObjects WithTag()**函数是一个更好的选

择。在这里介绍Game Object.Find的目的主要是为了让读者知道这个函数的存在，在有些时候需要使用这个函数来查找一些没有使用特定Tag的对象。

- 除了删除LevelTimer对象，OnDestroy()函数还会查找场景中所有的焰火对象，并启动这些对象。OnDestroy()函数中调用了GameObject.FindGameObjectsWithTag()函数来查找所有具有匹配Tag的对象。GameObject.FindGameObjectsWithTag()函数会返回所有具有“Firework”标签的对象，并通过调用Play函数启动烟火系统中的每一个对象。



如前所述，Unity中的每一个对象都是一些相关组件的集合。例如，一个标准立方体（使用菜单GameObject | 3D Object | Cube所创建的），就是由Transform组件、Mesh Filter组件、Mesh Renderer组件以及Box Collider组件所组成的。这些组件共同决定了立方体的性质。

在脚本中使用函数GetComponent()可以获得一个目标组件的引用，这样就可以直接访问目标组件的公共属性。前面代码中的OnDestroy()函数就使用GetComponent()函数获得了“Particle System”组件的引用，这个函数是一个十分重要而且有用的函数，关于GetComponent()函数的更多详细信息，可以访问<http://docs.unity3d.com/Script Reference/GameObject.GetComponent.html>上的Unity在线文档。

## 2.8 游戏测试

至此，已经完成第一个Unity游戏了！现在是时候来运行一下这个游戏了。Unity中游戏的测试过程首先是单击工具栏上的“Play”按钮，

然后以游戏者的视觉角度来进行游戏，以此来查看该游戏能否正常工作。除了进行游戏以外，还可以使用调试模式，在整个游戏进行过程中时刻观察着对象Inspector面板处所有公共和私有变量值的变化，以此来保证在游戏过程中所有变量的值都不会超出正常范围。激活调试模式的方法是，单击对象检查（Inspector）面板右上角的菜单图标，并从出现的上下文菜单中选中“Debug”选项，如图2.32所示。

图2.32 从检查（Inspector）面板处激活调试模式

激活调试模式以后，检查（Inspector）面板的变量和组件的外观可能发生改变，通常情况下，可以一个更详细和更准确的方式来查看这些变量，另外，还可以看到所有的私有变量。图2.33所示的就是在调试模式下看到的变换（Transform）组件。

图2.33 在调试（Debug）模式下查看到的变换（Transform）组件

在游戏运行时，另一个十分有用的工具是Stats面板，可以通过在工具栏上单击Stats按钮来将游戏（Game）选项卡切换到统计（Stats）选项卡，如图2.34所示。

图2.34 从游戏（Game）选项卡切换到统计（Stats）选项卡

游戏（Game）选项卡只有在Play模式下才可以使用，在这种模式下，它会详细地显示游戏中的所有关键性能数据统计，例如帧速率

（FPS）和内存使用情况。这些数据可以诊断任何可能会影响到游戏的问题。FPS表示每秒中帧（时间单元或者周期）的数量，当然这里并没有办法说什么值是好的FPS值，什么值是差的FPS值，或者某个值是最为合适的FPS值。但是通常认为较高的FPS值要好于较低的FPS值，这是因为较高的FPS值意味着游戏在一秒完成了更多的周期。如果FPS值小于20甚至15，那么很有可能是游戏出现了延迟，因此要花费更多的时间来完成一个周期。很多游戏内部和外部的变量都会影响到FPS。内部因素包括场景中光源的数量、网格对象的顶点密度、指令的数目、代码的复杂度。外部因素包括计算机硬件配置、当前正在运行的程序和进程的数量及硬盘空间的大小等。

简而言之，如果FPS值比较低，那么就意味着在游戏中存在一些需要注意的问题。这个问题的解决需要考虑到很多方面，例如网格是不是过于复杂，它们上面是否有过多的顶点，贴图是不是太大了，游戏中是不是加入了太多的音效等。图2.35所示为一个正常运行的金币采集游戏示例图，这个完整的游戏可以在本书的配套文件中的“Chapter02/End”文件夹中找到。

图2.35 对金币采集游戏进行测试

## 2.9 构建

现在是时候来构建（Build）游戏了，构建（Build）是指将游戏进行编译和打包，使得游戏成为一个独立的，可以自执行的形式，玩家在没有Unity编辑器的情形下就可以运行这个游戏。通常情况下，在开

发游戏时，就应该考虑到这个游戏的运行环境（例如Windows、iOS、Android及其他），要知道这是在设计阶段就应该决定的，而不是在开发结束时才想到的问题。虽然我们经常听说Unity是一种“一次开发，随处运行”的工具，但其实这个口号可能会带来负面的效果，尽管它宣称在游戏做好了之后，就可以像在桌面系统上一样轻松地在任何平台上运行这个游戏。

不幸的是，问题并非如此简单，能够在桌面系统运行良好的游戏，不一定能在手机上完美运行。反之亦然，主要是由于这些系统硬件之间和行业标准之间存在着巨大差异。考虑到这些差异，我们把主要精力放在Windows和Mac等桌面平台上，暂时不考虑手机和游戏机等平台。如果想创建一个在桌面平台上运行的游戏，可以在菜单栏上依次选择“File | Build Settings”，如图2.36所示。

图2.36 选择“Build Settings”

之后，“Build Settings”对话框就会显示出来，这个界面中包含了3个主要的区域，“Scenes In Build”区域中列出了在进行游戏构建时会包括的所有场景，注意是所有的场景，并不考虑这个场景是否真的可以被玩家访问。总之，如果在游戏中需要一个场景，那么这个场景必须出现在列表中，最开始的时候，这个列表是空的，如图2.37所示。

图2.37 “Build Settings”对话框



可以从项目（Project）面板上将场景资源拖曳到“Scenes In Build”列表中，这样就可以轻松地将场景添加进来。对于金币采集游戏，需要将“Level\_01 scene”添加到列表中。当场景添加进来以后，Unity中就会自动为其分配一个编号，这个编号的值取决于它在列表中的顺序。0代表最高的场景，1代表紧随其后的那个场景，以此类推。最高的场景就是开始的场景，也就是说，当游戏在生成之后运行时，会从编号为0的scene开始。因此，scene0通常也就是游戏的序篇或者介绍场景，如图2.38所示，在“Build Settings”对话框中添加了一个关卡。

图2.38 向“Build Settings”中添加一个关卡

接下来，在“Build Settings”对话框的左侧的“Platform”列表框中选择游戏运行的目标平台。对于桌面系统来说，应该选择“PC, Mac & Linux Standalone”，这也是默认的选项，然后在右侧的“Target Platform”下拉列表框中选择Windows、Linux，或者Mac OS X中的一项，这要取决于所使用的系统，如图2.39所示。

图2.39 选中一个目标平台

如果之前在多个平台上对游戏进行过测试，或者在其他的平台例如Android和iOS上进行过测试，就可以使用左下角的“Switch Platform”按钮。当在Platform中选择一个选项之后，这个按钮就有可能被激活了。如果这个按钮被激活了，那么可以单击“Switch

Platform”按钮来向Unity确认，如图2.40所示，Unity会花费一点时间将来将资源配制成可以在选中平台上运行的程序。

图2.40 切换平台

在首次进行构建之前，想要查看一些与玩家相关的重要构建参数，例如游戏分辨率、游戏质量设置、可执行文件图标、相关信息以及其他设置，可以在“Build”对话框中选中“Player Settings”对游戏玩家设置进行调整。现在已经在对象检查（Inspector）面板中显示“Player Settings”了。同样，也可以在菜单栏上选中“Edit | Project Settings | Player”来完成这个操作，如图2.41所示。

图2.41 访问“Player Settings”选项

在“Player Settings”选项中，可以设置“Company Name”和“Product Name”这两个参数，它们将会成为内置在这个可执行文件中的信息。同样如果需要，还可以为这个游戏指定一个图标，为鼠标光标指定一个图像。对于这个采集游戏，后面的两项设置将不进行设定，如图2.42所示。

图2.42 为游戏设置厂商名字和产品名字

“Resolution and Presentation”选项卡也是十分重要的，因为在这里可以设置游戏屏幕的大小以及在游戏启动的时候是否同时出现分辨率

(Resolution) 对话框。在这个选项卡中，可以看到“Default Is Full Screen”选项已经处于被选中状态，这意味着游戏将会以一个全屏幕的形式运行，而不是在一个较小的可移动的窗口中运行。此外，将“Display Resolution Dialog”下拉列表框设置为“Enabled”，如图2.43所示。设置完成后，在游戏开始时就会出现一个选项屏幕，允许用户对目标分辨率和屏幕大小以及一些自定义控制进行选择。如果构建最终版本，则需要禁用这个选项，按照自定义的屏幕选项设置进行游戏。不过，对于构建测试版本，分辨率对话框是一个十分有用的功能，它可以实现在各种不同大小的屏幕中测试游戏。

图2.43 启用“Resolution”对话框

现在已经为第一个游戏的编译构建做好了准备。接下来，单击“Build Settings”对话框中的“Build”按钮，或者从应用程序菜单上依次选择“File | Build & Run”。之后Unity会弹出一个保存对话框，在这个保存对话框中需要指定生成文件在硬盘上的存储位置，选择一个位置，然后单击“Save”，构建的过程就完成了。有时这个过程会产生错误，错误信息以红色的字体显示在控制台窗口。这是有可能会发生的，例如，试图将文件保存到一个只读硬盘区域，或者空间不足，又或者没有在该计算机上的管理权限等。不过，如果游戏在编辑器中能正常运行，一般在构建过程中也都会成功，如图2.44所示。

图2.44 构建并运行这个游戏

当构建过程结束时，Unity会在指定位置创建一个新的文件。如果是Windows操作系统，产生的就是图2.45所示的一个可执行文件和一个文件夹。注意，这两者都是必要的，而且两者是相互依存的。如果希望其他人在不安装Unity的情况下就可以进行这个游戏，就需要将可执行文件和相关的文件夹都发给用户。

图2.45 Unity构建产生的文件

在游戏运行中，“Resolution”对话框将会显示，这里假设已经在“Player Settings”中将“Display Resolution Dialog”选项设置为“Enabled”（见图2.46）。用户可以选择游戏分辨率、游戏质量、输出显示器以及对玩家控制进行配置。

图2.46 从Resolution对话框做好游戏的准备

当单击“Play”按钮时，游戏就会在默认的全屏模式下开始运行了。游戏现在已经完成了，而且构建成功了。现在可以将这个游戏发给自己的朋友和家人们进行测试了！效果如图2.47所示。

图2.47 在全屏模式下运行金币采集游戏

当不想再玩这个游戏时如何才能退出呢？现在这个游戏中并没有Quit按钮或者主菜单。在Windows操作系统中，需要按下键盘上的“Alt

+ F4”组合键。如果是在Mac系统中，可以按下“Cmd + Q”组合键。如果是Ubuntu，按下“Ctrl + Q”组合键。

## 2.10 小结

到现在为止，已经完成了第一个Unity工程——金币采集游戏。读者已经见识了Unity中大量的特性和功能，包括关卡的编辑与设计、预设体、粒子系统、网格、组件、脚本文件以及构建设置。当然这些特性和功能还需要进一步进行扩展和深入，现在已经将这些内容融合到了金币采集游戏中。接下来，需要继续一个完全不同的游戏，在这个游戏中会再次使用到这些功能和特性，并且将介绍一些全新的功能。简而言之，下一章，我们将会从一个Unity的初学者过渡成为一个Unity中等程度的使用者。

## 第3章 太空射击游戏 (I)

本章将会进入到一个新的学习领域，要开发的第二个游戏是一个使用双轴（Twin-stick）模式的太空射击游戏。双轴（Twin-stick）模式的游戏是指游戏玩家通过两个维度或者轴的输入来控制运动范围，一个轴负责运动，另一个轴负责角度。《僵尸邻居》（Zombies Ate My Neighbors）和《几何战争》（Geometry Wars）都是典型的双轴（Twin-stick）模式的游戏。游戏的功能主要是依靠C#语言进行编码来实现的。这么做的目的是为了展示如何通过脚本来实现一个游戏，即使在不使用编辑器和关卡构建工具的情况下。当然仍然会在一定情况下使用这些工具，但是不会很多，这是有意为之的。现在假设读者已经完成了前面两章中创建的游戏项目，同时也具有优秀的C#编程基础，虽然C#并不是学习Unity 3D必备的知识。现在开始着手双轴（Twin-stick）模式的太空射击游戏。本章将会包括以下重要主题：

- 创建操作与预设体
- 双轴控制以及轴向运动
- 玩家角色控制器与射击机制
- 基本的敌人运动与人工智能（AI）



牢记下面的这个例子以及相关的工作，抽象地讲，这里面涉及的工具和概念在很多应用程序中都是通用的。可能读者想要创建的并不是双轴的



射击游戏，不过没关系，实际上本书也不可能将所有想要做的游戏类型都介绍一遍。重要的是，通过书中的范例，向读者传达游戏开发的思路 and 工具的使用，这些内容都可以再次应用在游戏中。理解这一切才是学习Unity或者其他开发工具最为重要的。

## 3.1 对完整的项目进行展望

在开始开发双轴（Twin-stick）模式的太空射击游戏之前，先来看看完成之后的项目是什么样子以及它是如何工作的，如图3.1所示。要开发的这款游戏只包含一个场景，在这个场景中，玩家可以控制一个宇宙飞船来射击迎面而来的敌人，通过键盘上的方向键或者“W”“S”“A”“D”键来控制宇宙飞船在关卡中移动，而且宇宙飞船的方向始终跟随着鼠标。单击鼠标左键来控制宇宙飞船开火。

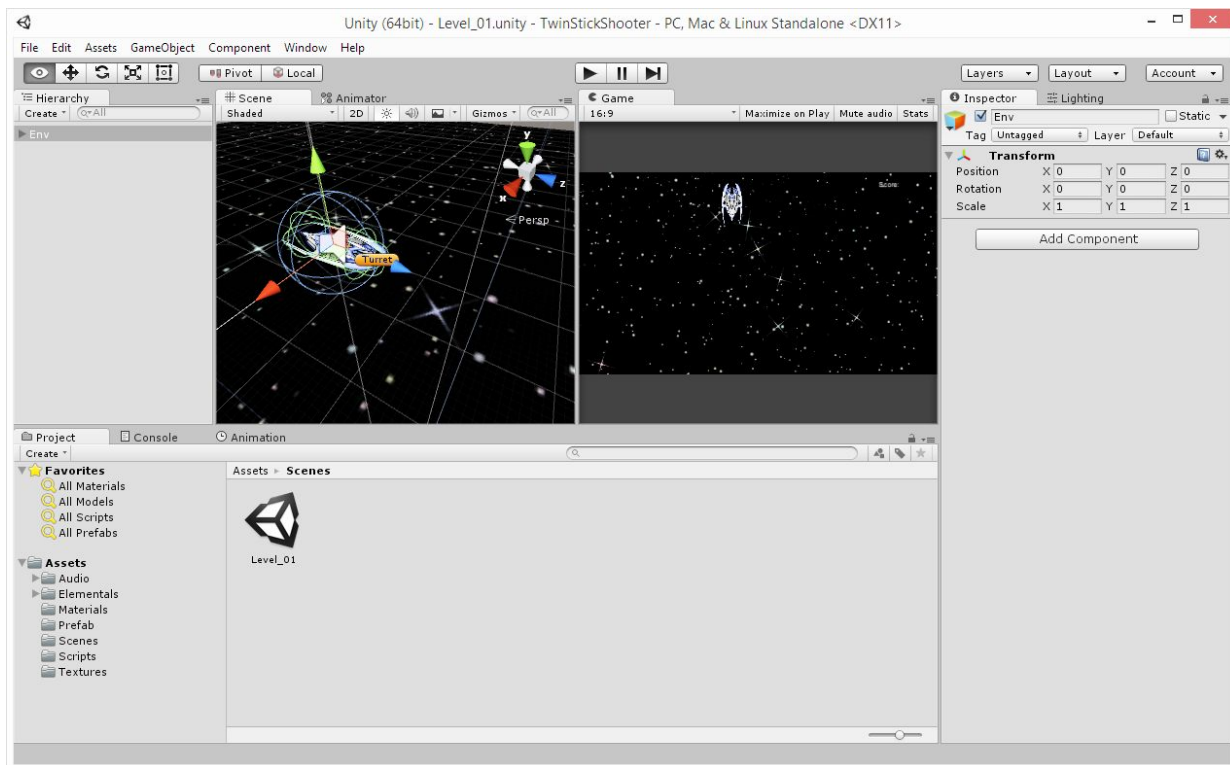


图3.1 完成的双轴（Twin-stick）模式的太空射击游戏



本章和下一章中将对这个项目进行讨论，该项目的完整文件 TwinStickShooter 可以在本书配套文件的“Chapter03/ TwinStickShooter”文件夹中找到。

这个项目中使用的资源（包括音效和贴图）都是从 OpenGameArt.org 网站上免费下载的，可以在这个网站上找到很多的游戏资源，可以通过公共领域、普通许可或其他许可来免费使用这些资源。

## 3.2 开始太空射击游戏

首先，创建一个不包含任何包和资源的空白 Unity 3D 项目。创建一个新项目的细节可以参考第1章。这一次的项目要从头开始，当项目成功创建之后，创建一些基本的文件夹来组织项目中要使用的资源，这一点在管理文件时是相当重要的。为贴图、场景、材质、声音、预设体以及脚本创建对应的文件夹，如图3.2所示。

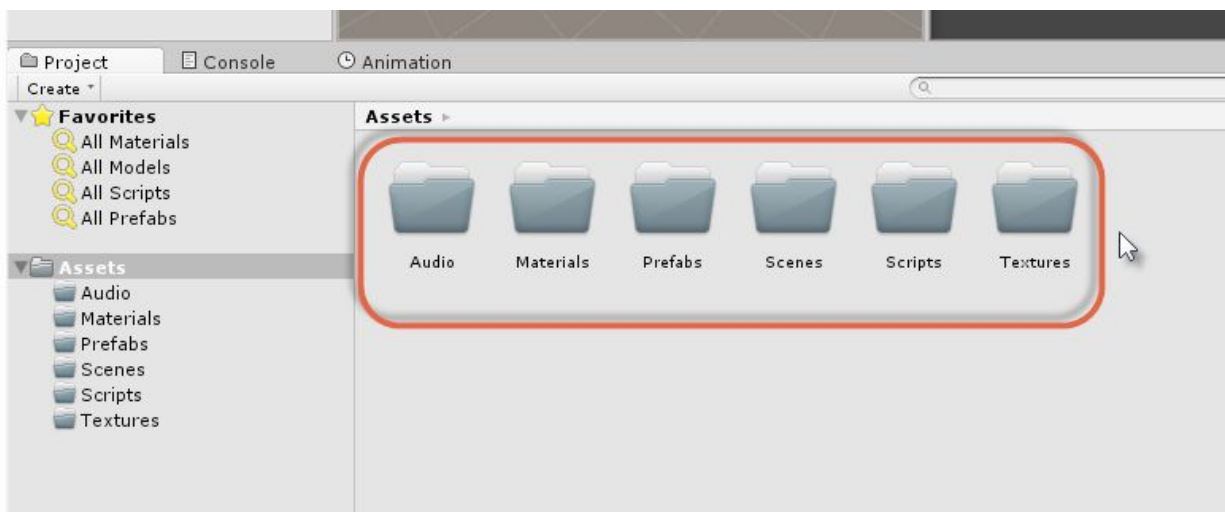


图3.2 创建文件夹来实现资源的组织

游戏中还需要一些图形和音频资源。可以在本书配套文件的“Chapter03/Assets”文件夹中，或者从OpenGameArt.org网站中找到这些资源。现在从玩家的宇宙飞船、敌人的宇宙飞船和星空背景的贴图开始导入。首先从Windows的资源管理器或者Mac的finder中将贴图拖曳到Unity中的项目（Project）面板里的Textures文件夹中。Unity就会自动地完成贴图的导入和配置工作，如图3.3所示。

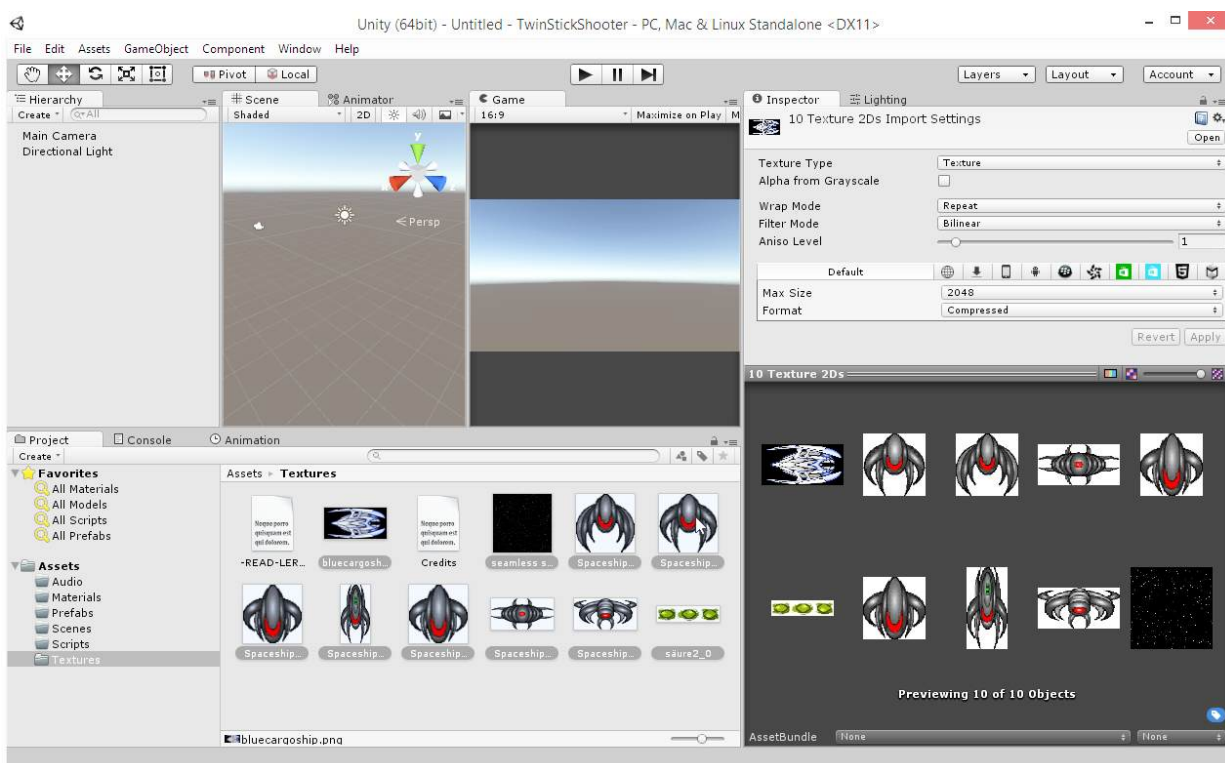


图3.3 导入玩家宇宙飞船、敌人宇宙飞船、星空背景和弹药的贴图



这些提供的资源可以使用，也可以不使用。如果读者愿意，完全可以创建自定义的资源，只需要将这些自定义的资源拖放到对应的资源文件夹

中，然后按照下面的教程继续操作。

默认情况下，Unity中导入图像文件作为3D对象的贴图，这些图像的像素大小都是2的倍数（例如4、8、16、32、64、128、256等）。如果导入图像的像素大小并不是以上数值中的一个，Unity就会对这些图像进行放大或者缩小到最近似的有效大小值。这并不是一个最适当的做法。对于一个2D的自上而下的空间射击游戏来说，导入的图片需要以其没有任何修改的最初大小出现。可以在对象检查（Inspector）面板中将所有导入的贴图选中，然后将它们的“Texture Type”属性由“Texture”改变为“Sprite (2D and UI)”。当完成改变之后，单击“Apply”按钮来更新设定，贴图将会保留其导入时的大小，如图3.4所示。

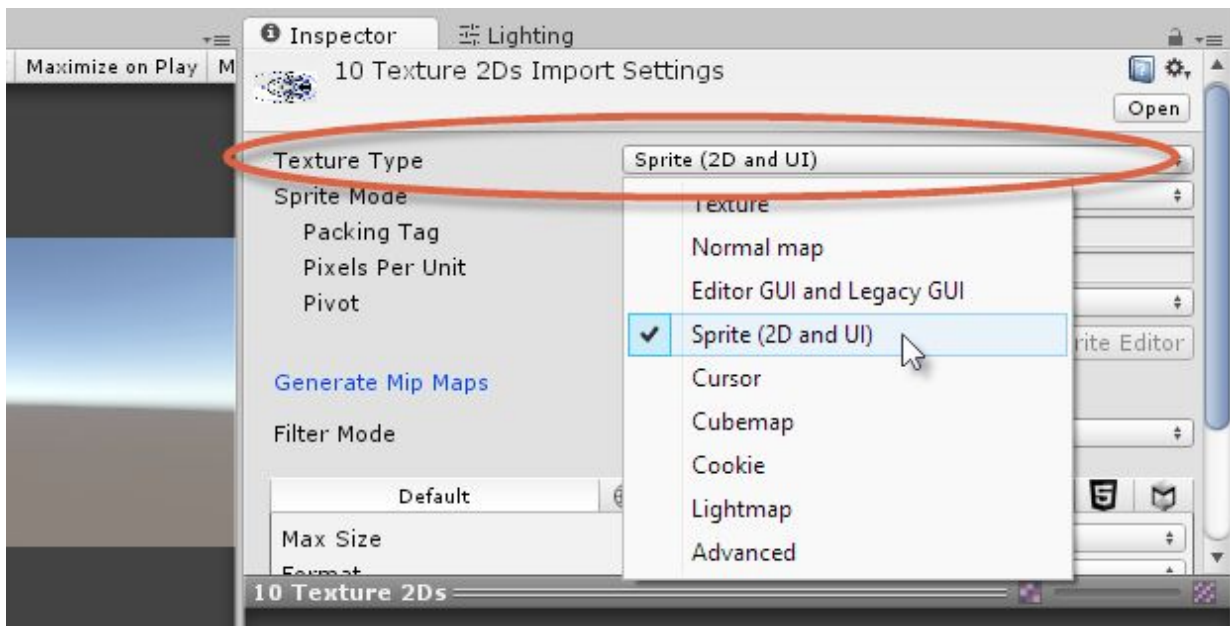


图3.4 为导入的图像修改“Texture type”属性

在将“Texture Type”的值设置为“Sprite (2D and UI)”之后，如果“Generate Mip Maps”复选框处于选中状态，就要将其取消。这样就可以避免Unity自动地根据摄像机的距离将贴图的画面质量降低。关于2D Texture和Mip Map的更多信息可以在Unity的在线文档<http://docs.Unity3d.com/Manual/class-TextureImporter.html>中找到，如图3.5所示。

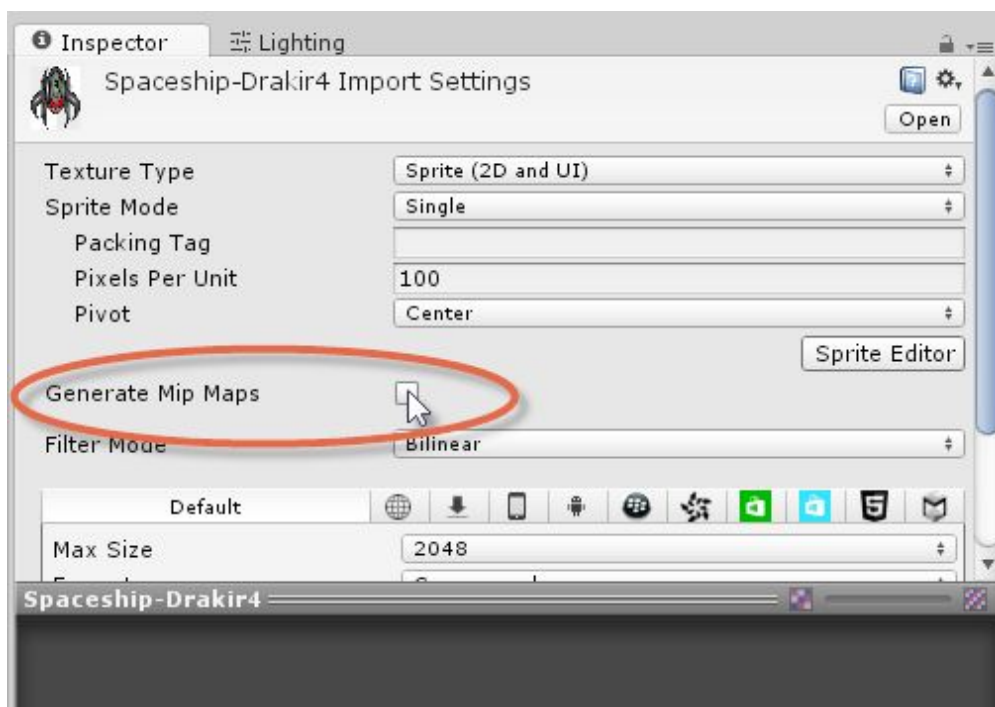


图3.5 取消“MipMapping”功能

现在可以将贴图以图片精灵（Sprite）对象的形式拖曳到场景中。不能将它们从项目（Project）面板拖曳到视图中，但是可以将它们从项目（Project）面板拖曳到层次（Hierarchy）面板上。当完成这个操作之后，贴图就会自动地以图片精灵对象的形式添加到场景中。在创建宇宙飞船对象时要经常使用这个功能，如图3.6所示。



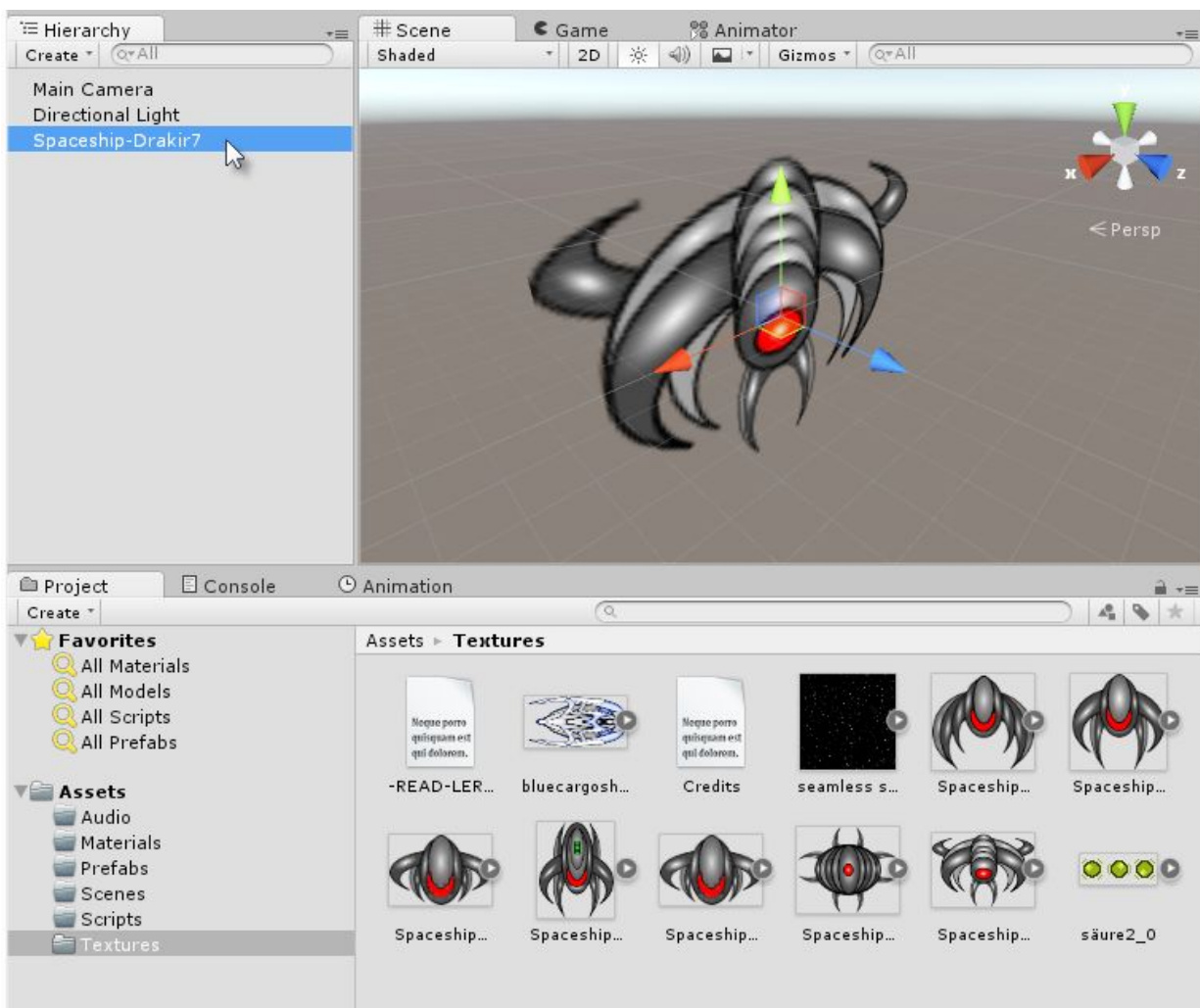


图3.6 向场景中添加图片精灵对象

接下来，导入游戏中需要的音乐和音效，这些内容可以在本书的配套文件中的“Chapter03/Assets/Audio”文件夹中找到。这些资源是从OpenGameArt.org下载的。可以简单地将这些文件从Windows的资源管理器或者Mac的Finder中拖曳到项目（Project）面板上。当完成这项操作之后，Unity中就自动地完成了资源的导入和配置。现在可以对导入的声音进行一个测试，测试的方法是：单击对象Inspector面板中的工具栏上的“Play”按钮，如图3.7所示。



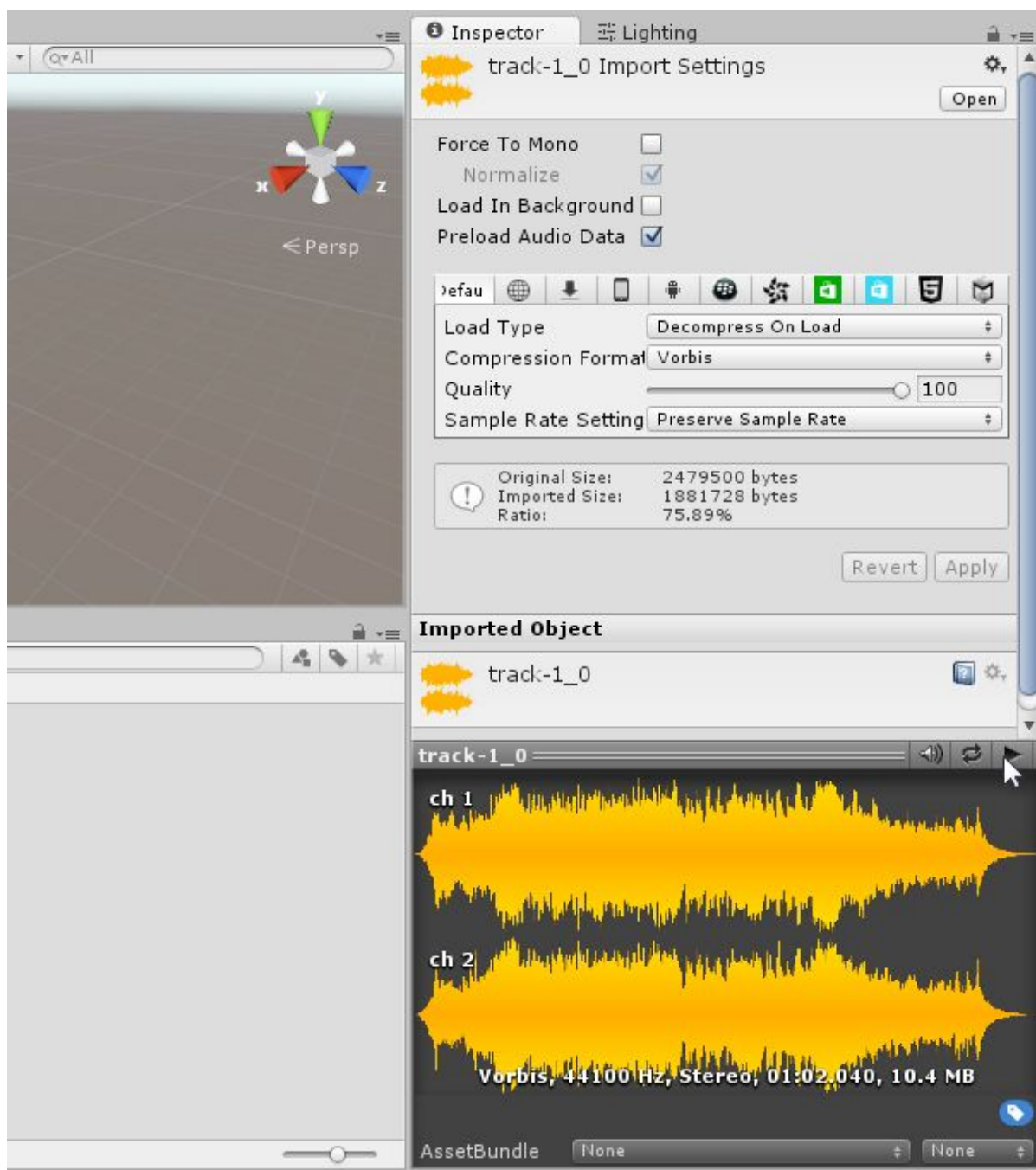


图3.7 从对象检查（Inspector）面板处对声音进行测试

跟导入贴图一样，Unity导入声音文件时也使用了一些默认的参数。这些参数十分适合那些时长很短的音效，例如脚步声、枪声和爆炸声。但是如果导入的是时长较长的声音，例如乐曲，这些参数就可

能出现问题，造成加载时间过长。为了修复这个问题，可以在项目（Project）面板选中音轨，然后在对象检查（Inspector）面板中，取消“Preload Audio Data”复选框的选中状态，最后在“Load Type”下拉列表框中选中“Streaming”选项，如图3.8所示。这样就可以确保音轨是流式传输的，而不需要在关卡启动时就完全加载到内存中。

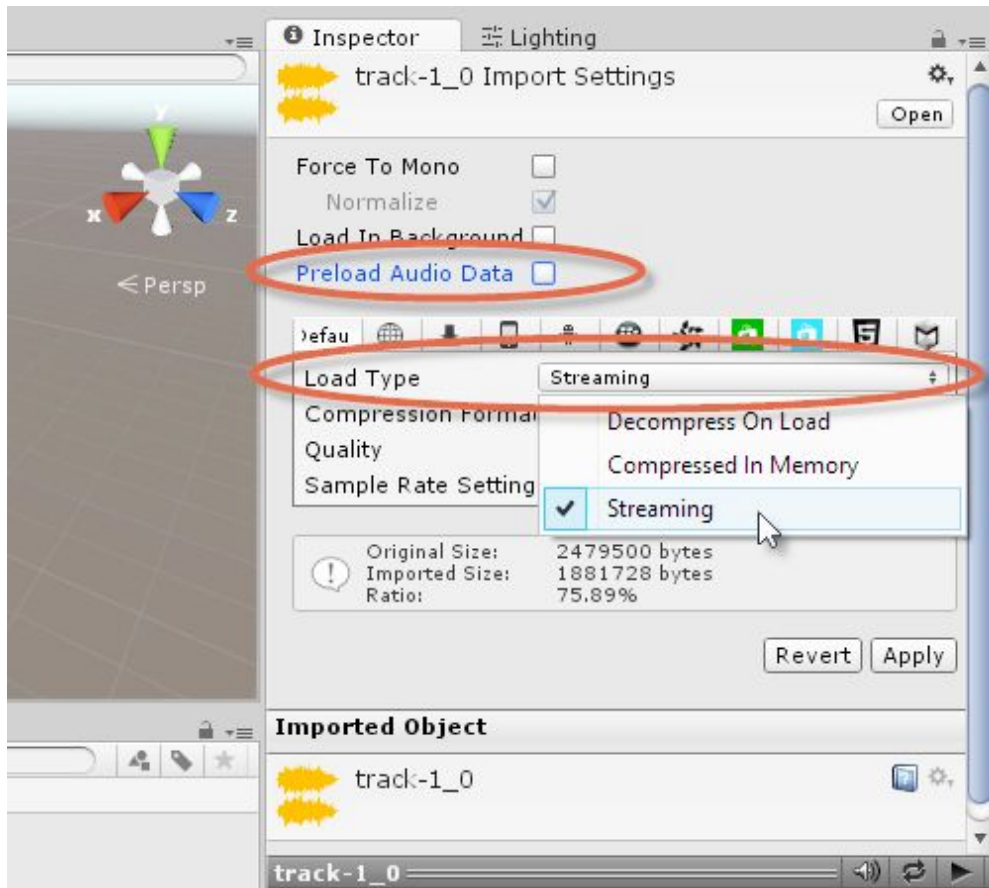


图3.8 将音轨设置为流式（Streaming）

### 3.3 创建玩家对象

现在已经为双轴模式射击游戏导入了所有需要的资源，也做好了创建一个玩家宇宙飞船对象的准备。玩家宇宙飞船对象就是指受玩家

控制而移动的游戏对象。现在创建一个对象可以说是毫无难度了，比如简单地将相关的玩家图片精灵从项目（**Project**）面板拖动到场景中就可以完成这个工作，但其实事情并非如此。这个游戏中的玩家所要控制的对象功能十分复杂，后面内容将会很快介绍。基于这些复杂的功能，在创建这个玩家对象时就要花费一些心思了。首先，在菜单栏上依次单击“**GameObject | Create Empty**”在场景中创建一个空的游戏对象，然后将这个对象命名为“**Player**”，如图3.9所示。

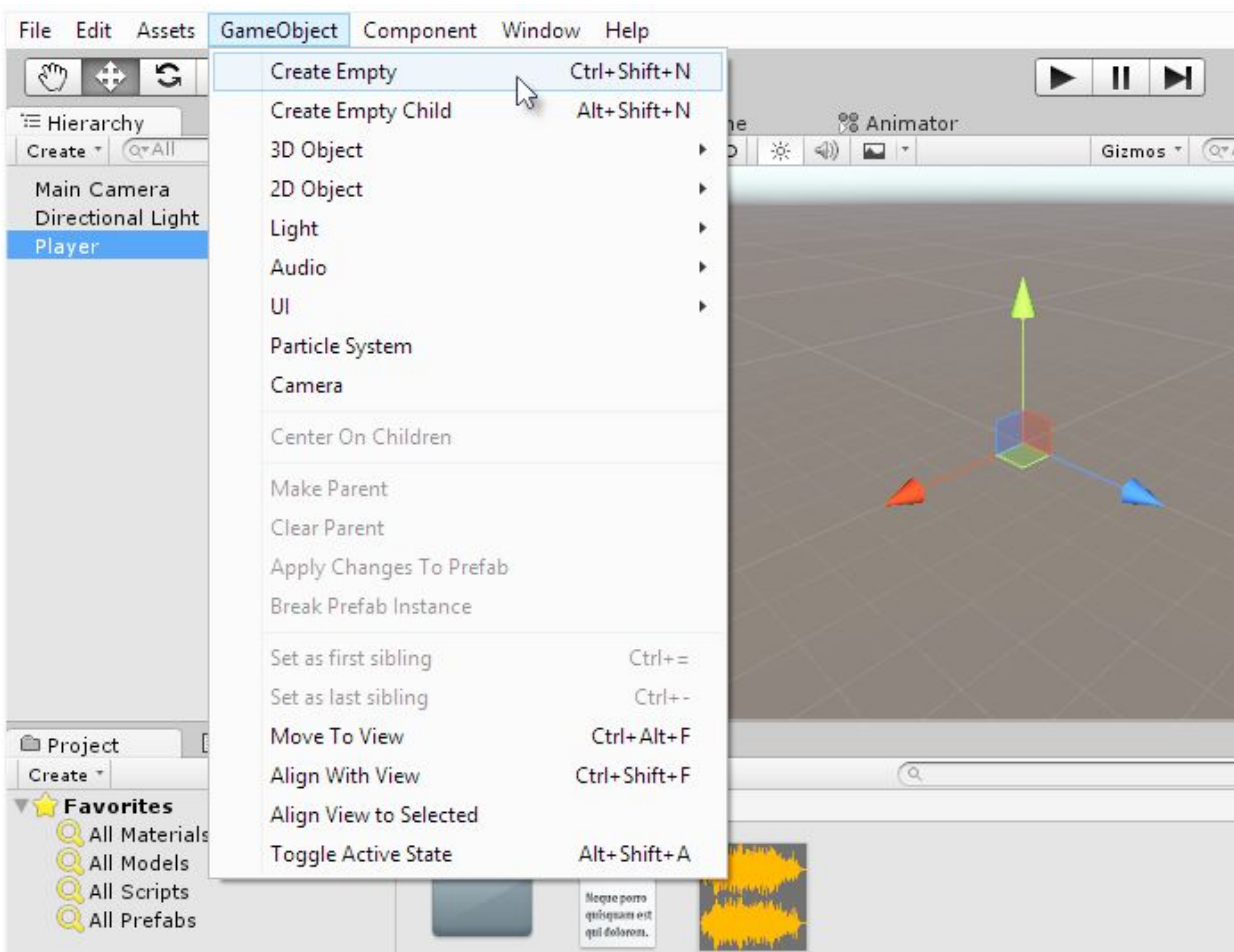


图3.9 创建一个玩家对象

新创建的游戏对象可能并不位于整个游戏世界的最中心（0, 0, 0），另外它的“Rotation”属性的值也可能不是（0, 0, 0）。为了确保新的游戏对象保持原状，可以在对象检查（Inspector）面板中将“Transform”组件的所有属性都设置为0。还有一种更为快捷的方式就是，单击“Transform”组件右侧齿轮状图标，然后选中上下文菜单中的“Reset”，如图3.10所示。

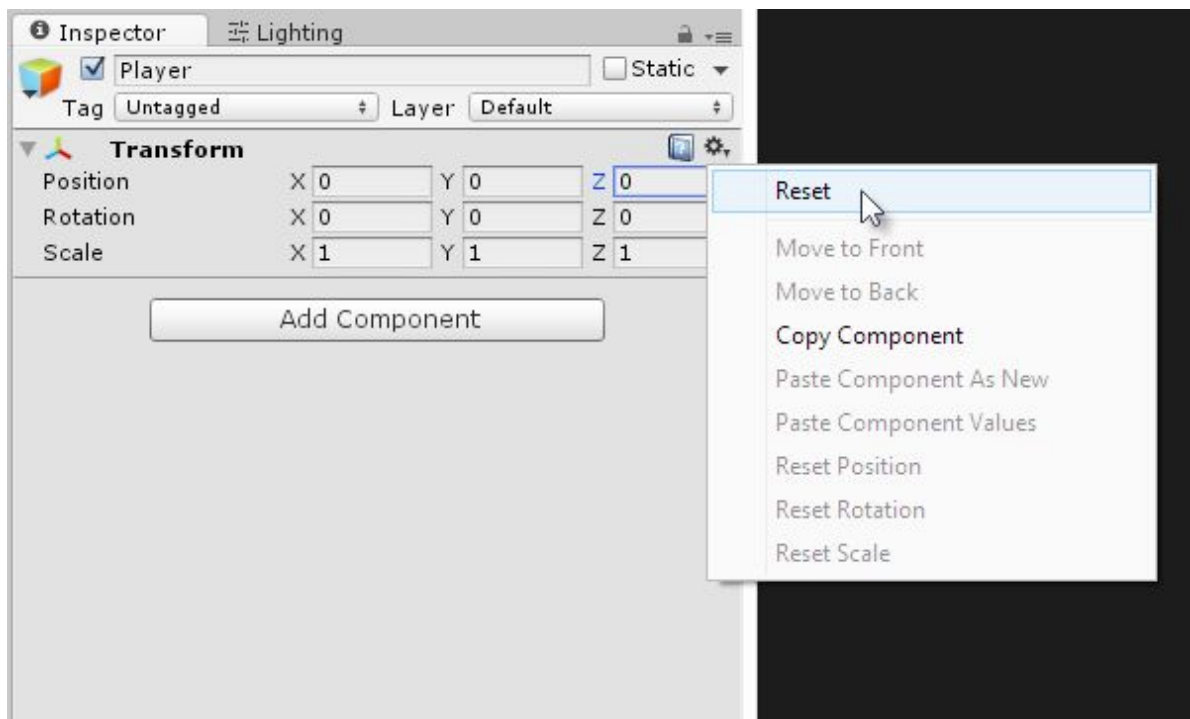


图3.10 重置“Transform”组件

接下来，将项目（Project）面板上的“drop ship”图片精灵（位于Textures文件夹中）拖动到层次（Hierarchy）面板，使它成为空玩家对象的子对象。然后，将“drop ship”图片精灵对象沿着X轴和Y轴各自旋转90度（见图3.11）。这使得“drop ship”图片精灵对象保持了和父对象相同的方向，同样也停靠在地面上。游戏中的摄像机采用了自上而下的视角。

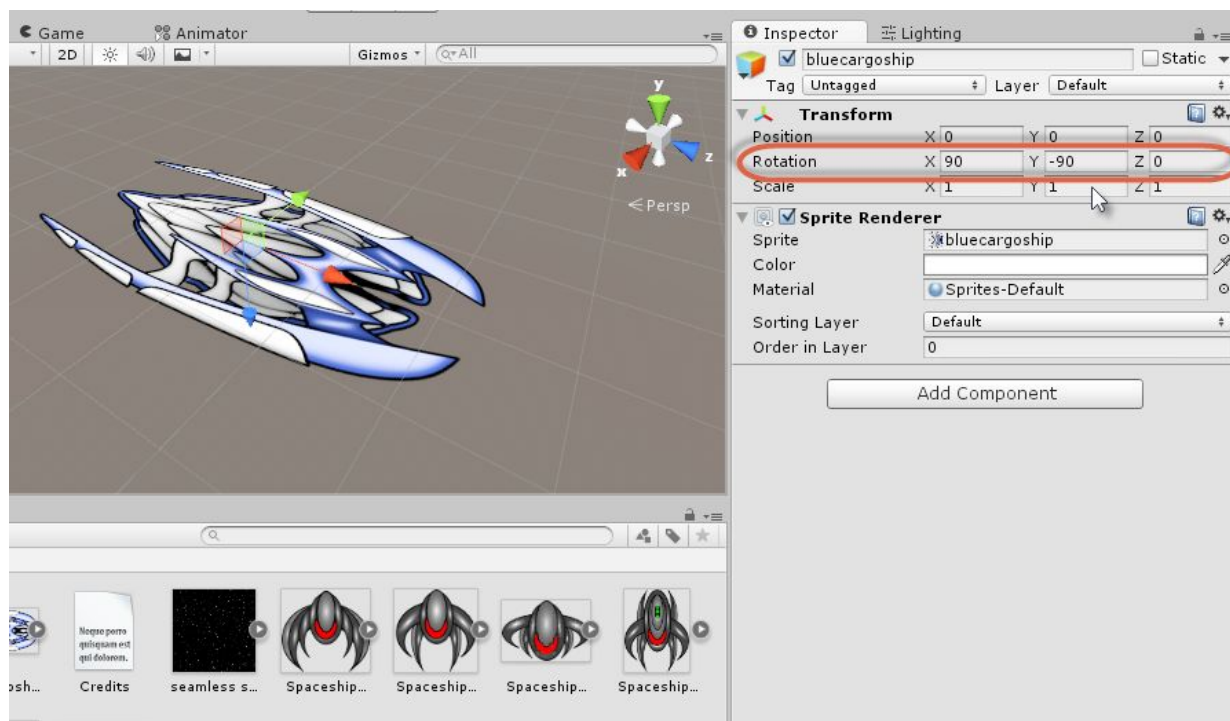


图3.11 调整玩家的宇宙飞船

现在可以选中玩家（**Player**）对象，并查看蓝色箭头的方向，以此来确认“ship”图片精灵对象已经和父对象是否摆放一致。现在“ship”图片精灵对象的方向应该和蓝色箭头的方向是相同的。如果不同，可以继续将“ship”图片精灵对象旋转90度直到它们对齐为止。在之后编写宇宙飞船航行的控制代码时，这一点十分重要，如图3.12所示。



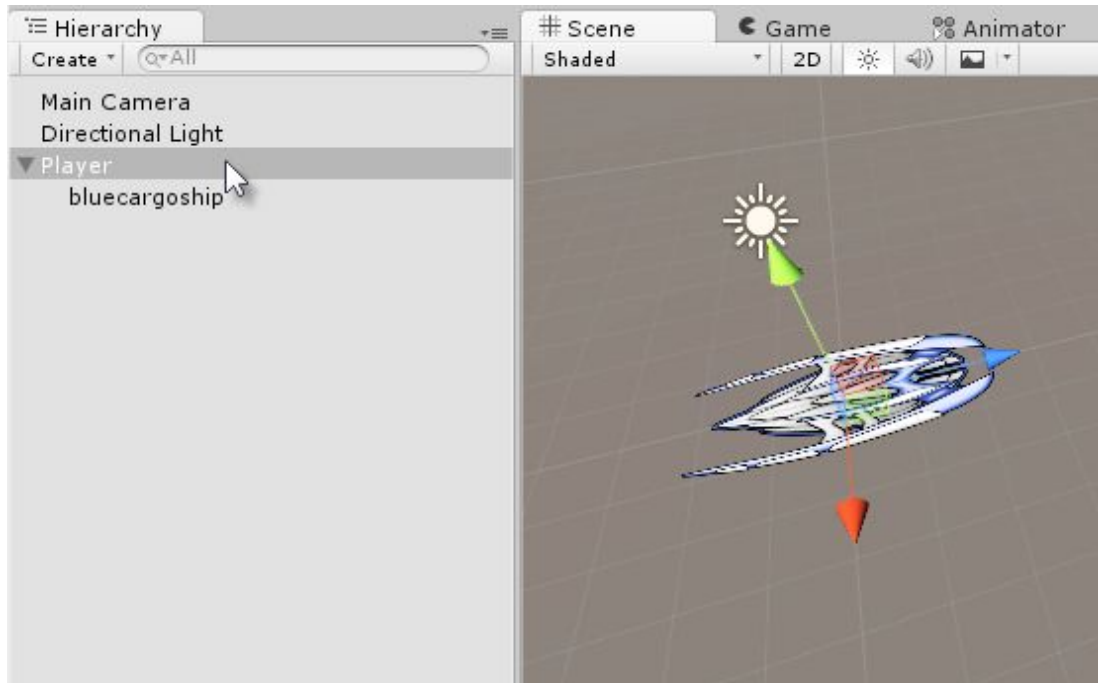


图3.12 把蓝色的小箭头称为“前向向量”

接下来，玩家（Player）应该具备现实世界的物理属性，也就是说玩家（Player）应该是固态的，而且会受到外力的影响。当玩家（Player）和其他固体接触时，就会发生碰撞；被敌人的武器击中时，也会造成损坏。为了实现这些功能，将两个额外的组件添加到玩家（Player）上，这两个组件就是刚体（Rigidbody）和碰撞体。首先选择玩家（Player）（注意不是图片精灵对象），然后在应用程序菜单上依次选中“Component | Physics | Rigidbody”，然后在应用程序菜单上依次选中“Component | Physics | Capsule Collider”，完成这两次操作之后，就为玩家（Player）添加了刚体和碰撞体组件，如图3.13所示。



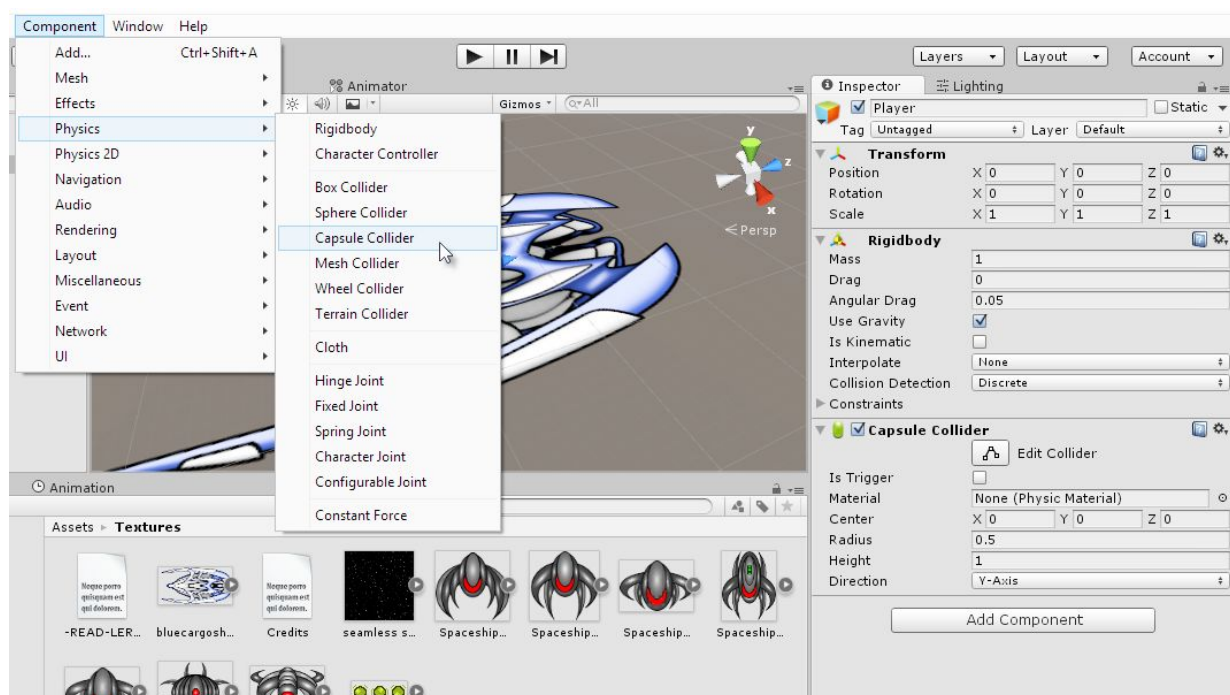


图3.13 向玩家（Player）添加刚体和碰撞体组件

碰撞体组件是用来模拟对象的体积的，刚体组件利用碰撞体来确定如何实际应用物理作用力。首先稍微对胶囊碰撞体（Capsule Collider）作出一点修正，因为默认的设置可能与玩家图片精灵不完全匹配。修改“Direction”“Radius”和“Height”的值，直到胶囊碰撞体与玩家图片精灵相匹配，而且与玩家对象的体积相同，如图3.14 所示。

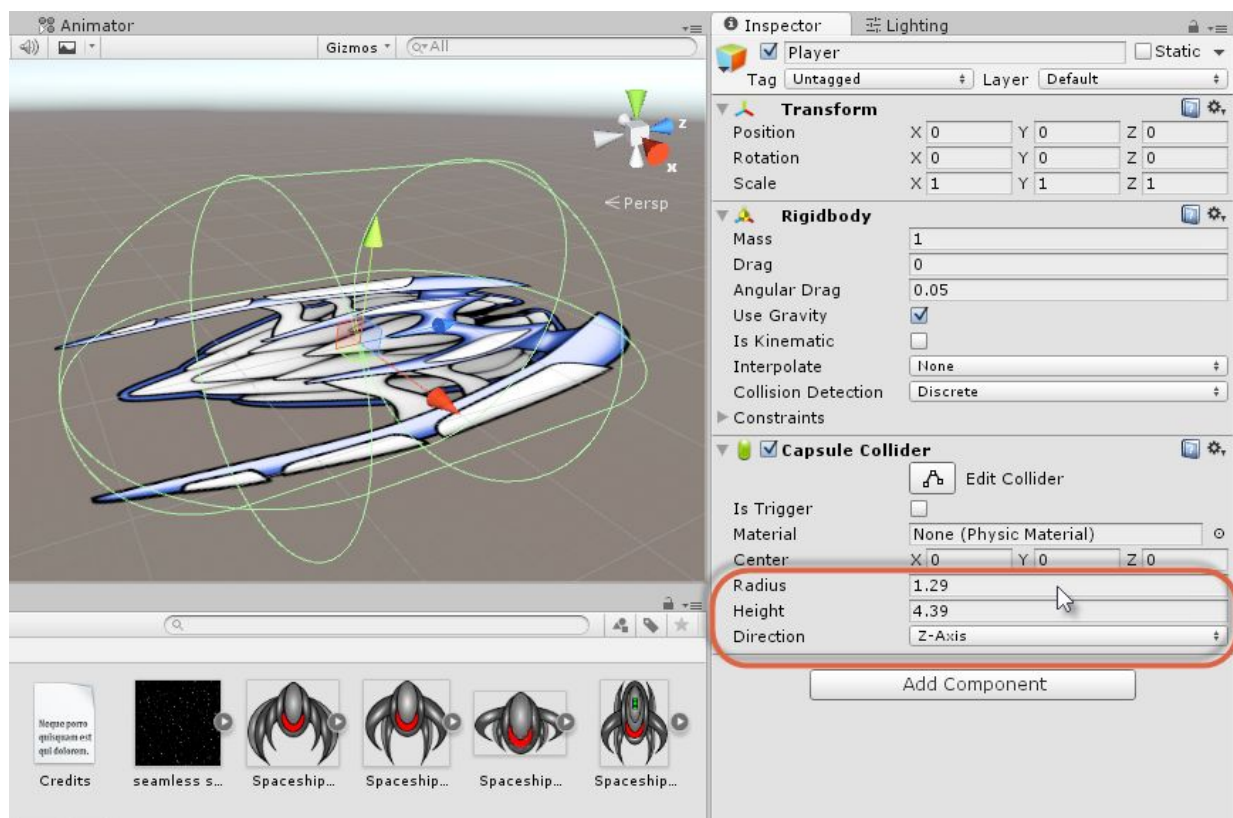


图3.14 对宇宙飞船的胶囊碰撞体进行调整

默认情况下，添加了刚体组件的对象如同现实中的物体一样，会受到重力的影响落到地面上。但是这并不适合游戏中会飞的宇宙飞船。因此，要对刚体（**Rigidbody**）组件进行修改。首先要取消“**Use Gravity**”单选框的选中状态，这样对象就不会再掉落在地上了。另外，在“**Freeze Position**”复选框中选中“**Y**”，然后在“**Freeze Rotation**”复选框中选中“**Z**”，这样就可以使宇宙飞船按照这款二维的自上而下游戏模式运行了，如图3.15所示。

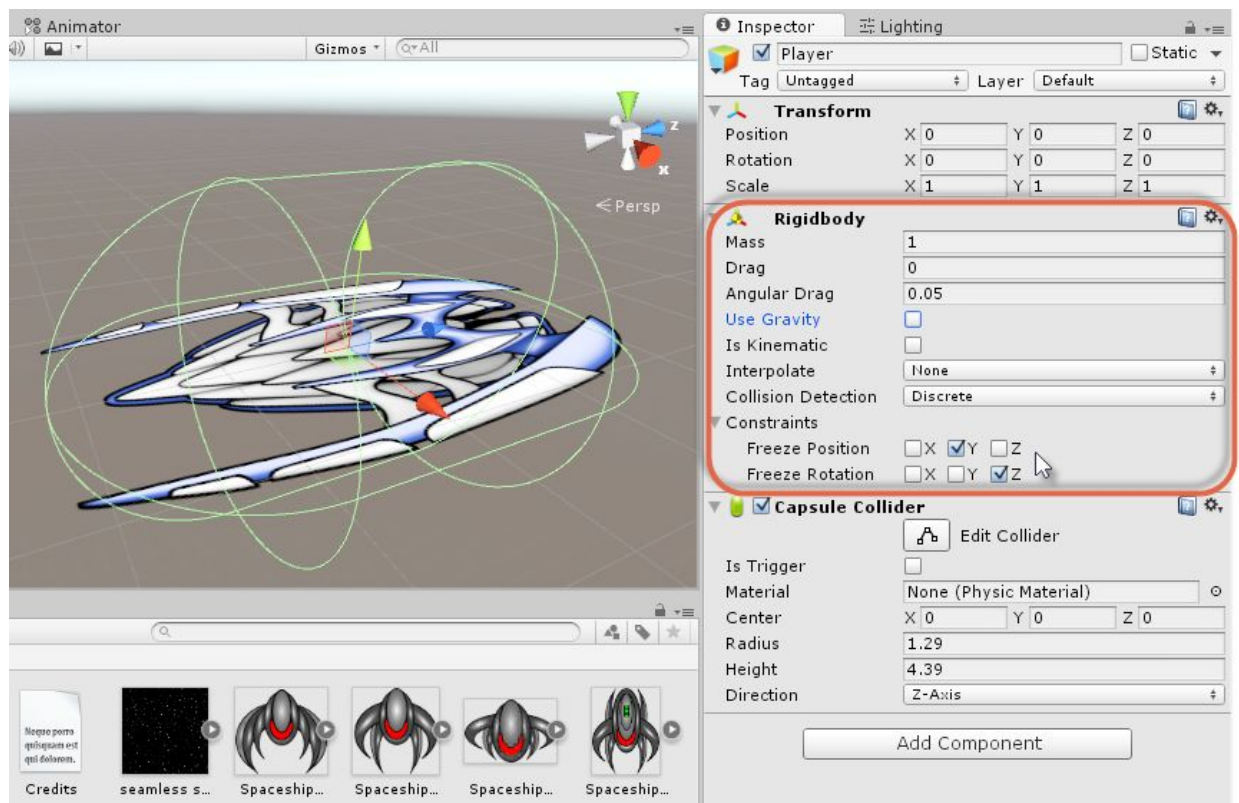


图3.15 为玩家宇宙飞船设置刚体（Rigidbody）组件

现在成功地设置了玩家宇宙飞船。当然，它暂时还不能在这个游戏中移动，是因为还没有为对象添加任何的代码，接下来将做这个工作，也就是玩家宇宙飞船可以响应用户的输入。

## 3.4 Player输入

现在已经在场景中创建了玩家（Player）对象，而且也为此对象设置了刚体（Rigidbody）和碰撞体（Collider）组件。但是，游戏对象并不会对玩家的任何输入进行响应。在一个双轴的游戏里，玩家通过两个轴的输入就可以完成游戏的控制，这通常意味着键盘上的“W”“A”“S”“D”4个键分别控制玩家的上、下、左、右。此外，鼠标

的移动控制玩家面向瞄准的方向，鼠标左键来控制武器。这些是游戏的控制方案。实现这个方案的操作步骤是：在项目（Project）面板上单击鼠标右键创建一个名为PlayerController.cs的C#脚本，如图3.16所示。

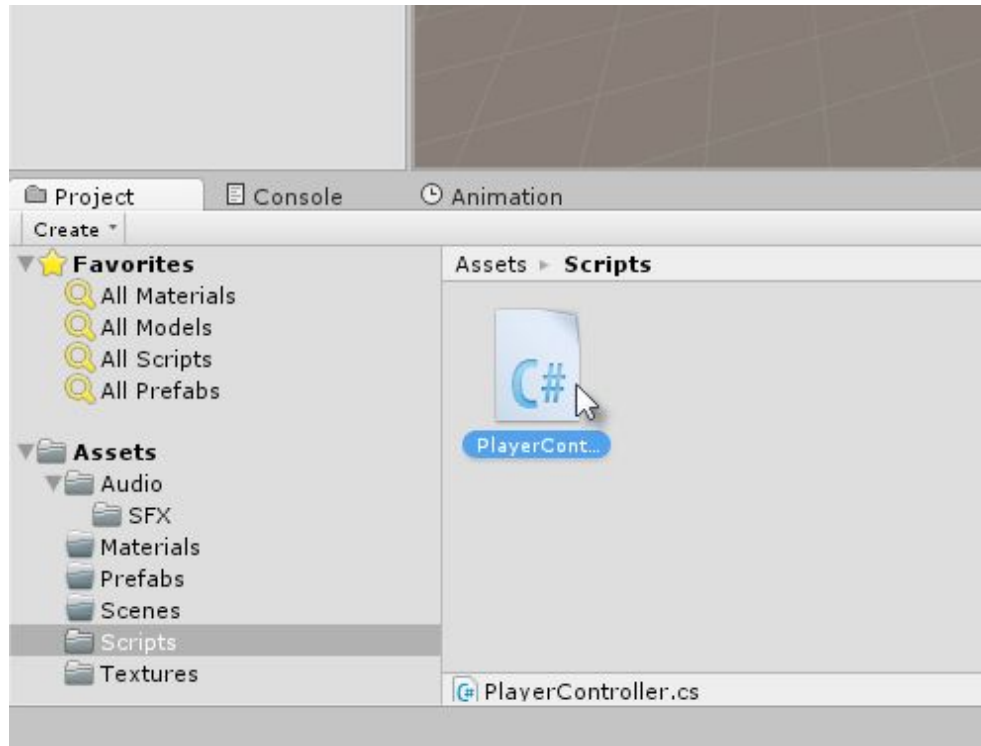


图3.16 创建一个玩家控制的C#脚本文件

在PlayerController.cs脚本文件中包含如下代码（如代码示例3.1所示）。

### 代码示例3.1:

```
//-----  
using UnityEngine;  
using System.Collections;  
//-----  
public class PlayerController : MonoBehaviour  
{  
    //-----  
    private Rigidbody ThisBody = null;
```

```

private Transform ThisTransform = null;

public bool MouseLook = true;
public string HorzAxis = "Horizontal";
public string VertAxis = "Vertical";
public string FireAxis = "Fire1";
public float MaxSpeed = 5f;

//-----
// 初始化函数
void Awake ()
{
    ThisBody = GetComponent();
    ThisTransform = GetComponent();
}
//-----
// 在每一帧都会调用Update()函数
void FixedUpdate ()
{
    //对运动进行更新
    float Horz = Input.GetAxis(HorzAxis);
    float Vert = Input.GetAxis(VertAxis);
    Vector3 MoveDirection = new Vector3(Horz, 0.0f, Vert);
    ThisBody.AddForce(MoveDirection.normalized * MaxSpeed);

    //对速度进行限制
    ThisBody.velocity = new Vector3
        (Mathf.Clamp(ThisBody.velocity.x, -MaxSpeed, MaxSpeed),
        Mathf.Clamp(ThisBody.velocity.y, -MaxSpeed, MaxSpeed),
        Mathf.Clamp(ThisBody.velocity.z, -MaxSpeed, MaxSpeed));

    //是否跟随鼠标
    if(MouseLook)
    {
        //改变角度 -视角跟随鼠标
        Vector3 MousePosWorld = Camera.main.ScreenToWorldPoint(new
            Vector3(Input.mousePosition.x,
            Input.mousePosition.y, 0.0f));
        MousePosWorld = new Vector3(MousePosWorld.x, 0.0f,
            MousePosWorld.z);
        //跟随光标的方向
        Vector3 LookDirection = MousePosWorld -
            ThisTransform.position;

        //FixedUpdate函数中实现旋转
        ThisTransform.localRotation = Quaternion.LookRotation
            (LookDirection.normalized,Vector3.up);
    }
}

```

```
}  
//-----
```

下面对代码示例3.1进行几点总结。

- **PlayerController**类需要依附在场景中的玩家（**Player**）对象上。因为这个脚本要接受玩家的输入，并控制宇宙飞船的移动。
- 当关卡开始后，创建对象的时候会调用**Awake()**函数，这个函数会查找“**Transform**”和“**Rigidbody**”（也就是刚体组件）两个组件，“**Transform**”组件用来控制玩家转动，“**Rigidbody**”组件用来控制玩家的运行。“**Transform**”组件可以通过“**Position**”属性来控制玩家的移动，但是这里并没有考虑到碰撞之类的情况。相比之下，“**Rigidbody**”组件可以防止玩家从其他的固体对象中穿过。
- 函数**FixedUpdate()**会在每次物理系统发生变化时调用，这个函数会在每秒都执行固定的次数。**FixedUpdate()**与**Update()**函数不同，**Update()**函数是每帧调用一次，但是因为帧速率是不断变化的，因此每秒钟执行的**Update()**函数的次数是不固定的。如果通过物理系统来控制一个对象，比如使用刚体（**Rigidbody**），就需要使用**FixedUpdate()**函数，而不是**Update()**函数。这是Unity中的惯例，最好牢牢地记住这一点。
- 在**FixedUpdate**函数执行时会调用**Input.GetAxis()**函数，它可以从键盘或者游戏手柄上读取轴向的输入数据。**Input.GetAxis()**函数会读取水平（左右）、垂直（上下）两个方向的数据。输入数据的值区间为-1到1，方向左键被按下时，水平轴的返回值为-1。方向右键被按下时，水平轴的返回值为1。当返回值为0时意味着这两个键都没有被按下，或者两个键都被按下了。垂直轴的原理相同，上意味着1，下意味着-1。如果没有键按下，就是0。可以在Unity



的在线文档<http://docs.Unity3d.com/>

[ScriptReference/Input.GetAxis.html](http://docs.Unity3d.com/ScriptReference/Input.GetAxis.html)中找到GetAxis()函数的更多信息。

- **Rigidbody.AddForce()**函数用来给Player对象施加一个力，将它向一个方向移动。“AddForce”也定义了一个速度，就是对象沿着特定方向运动的快慢。“MoveDirection”向量中定义了方向，这个向量是由垂直和水平两个轴共同决定的。这个方向上以最大的速度运行，就可以保证游戏对象走得最快。可以在Unity的在线文档<http://docs.Unity3d.com/ScriptReference/Rigidbody.AddForce.html>中找到AddForce()函数的更多信息。
- 函数**ScreenToWorldPoint**是用来将光标在游戏屏幕上的坐标转换为在游戏世界的坐标，这个光标的位置也就是游戏中玩家对象面向的位置。这段代码将会使玩家一直面向着鼠标光标。如果想要代码正确地工作，还需要进一步地调整。关于**ScreenToWorldPoint**函数的更多详细信息，可以访问Unity位于<http://docs.Unity3d.com/ScriptReference/Camera.ScreenToWorldPoint.html>的在线文档。

## 3.5 配置游戏中的摄像机

前面的代码已经实现了对玩家（**Player**）的控制，但是这段代码还不完美。其中一个问题就是玩家（**Player**）还不能一直面向着鼠标光标，虽然代码中设计了这样的功能。产生这个问题的原因在于摄像机，默认情况下的摄像机并不是按照自顶而下的2D游戏方案来设计的。现在需要调整摄像机的设置。首先选择位于场景（**Scene**）视图中

右上方的导航块（ViewCube），然后单击其中的向上的箭头，操作完成之后，玩家在游戏视角就切换成了由顶而下的视角，如图3.17所示。

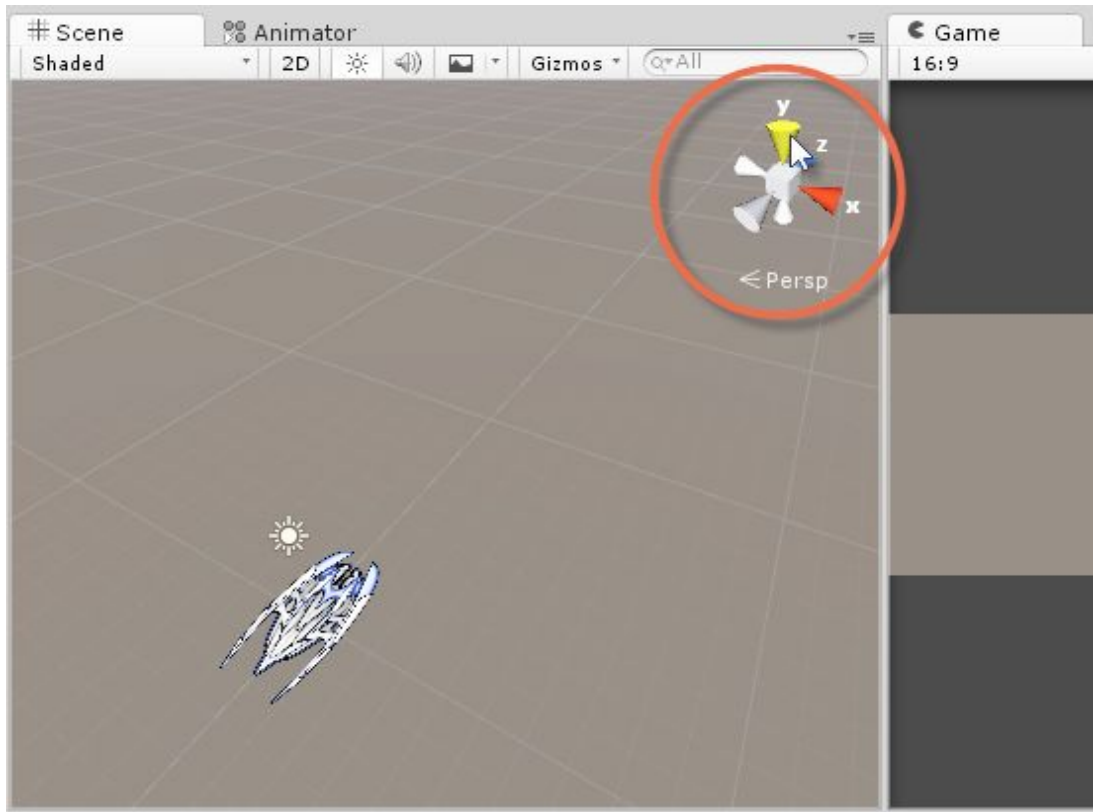


图3.17 使用导航块（ViewCube）可以切换游戏者的视角

现在已经实现了一个自顶而下的观察视角，这是因为导航块（ViewCube）已经将顶部设置为了当前的观察位置，如图3.18所示。

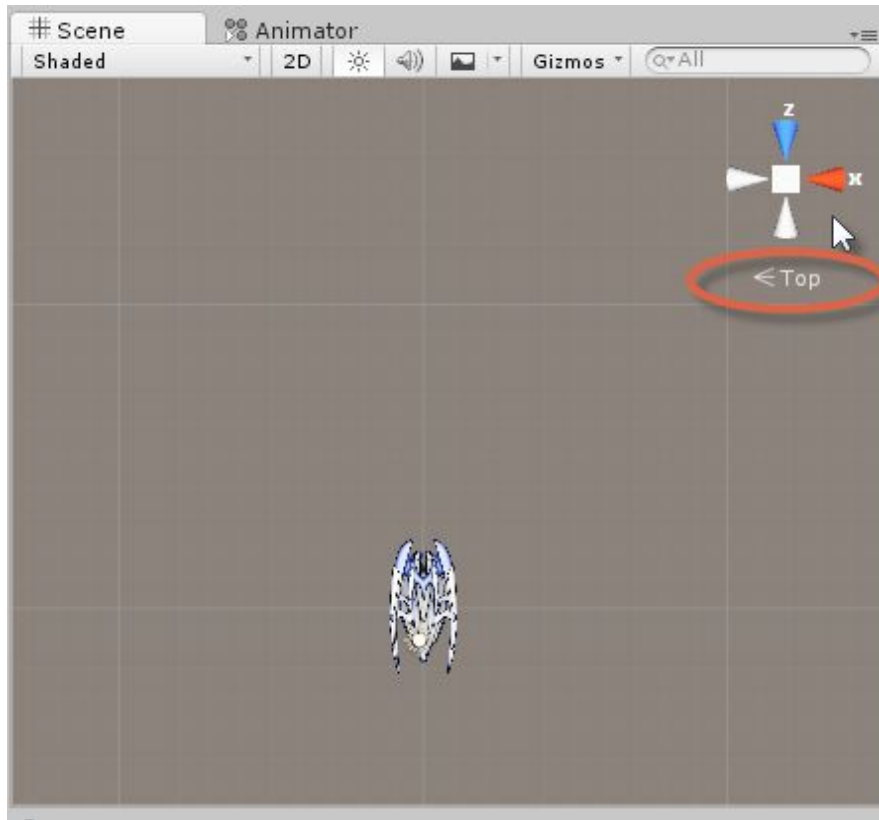


图3.18 自顶向下的观察视角

至此就有了一个符合游戏设计方案的摄像机，这个摄像机的视角是自上而下的。首先选中场景中的摄像机或者从层次（Hierarchy）面板中来选，然后在应用程序菜单中依次选中“GameObject | Align With View”，如图3.19所示。

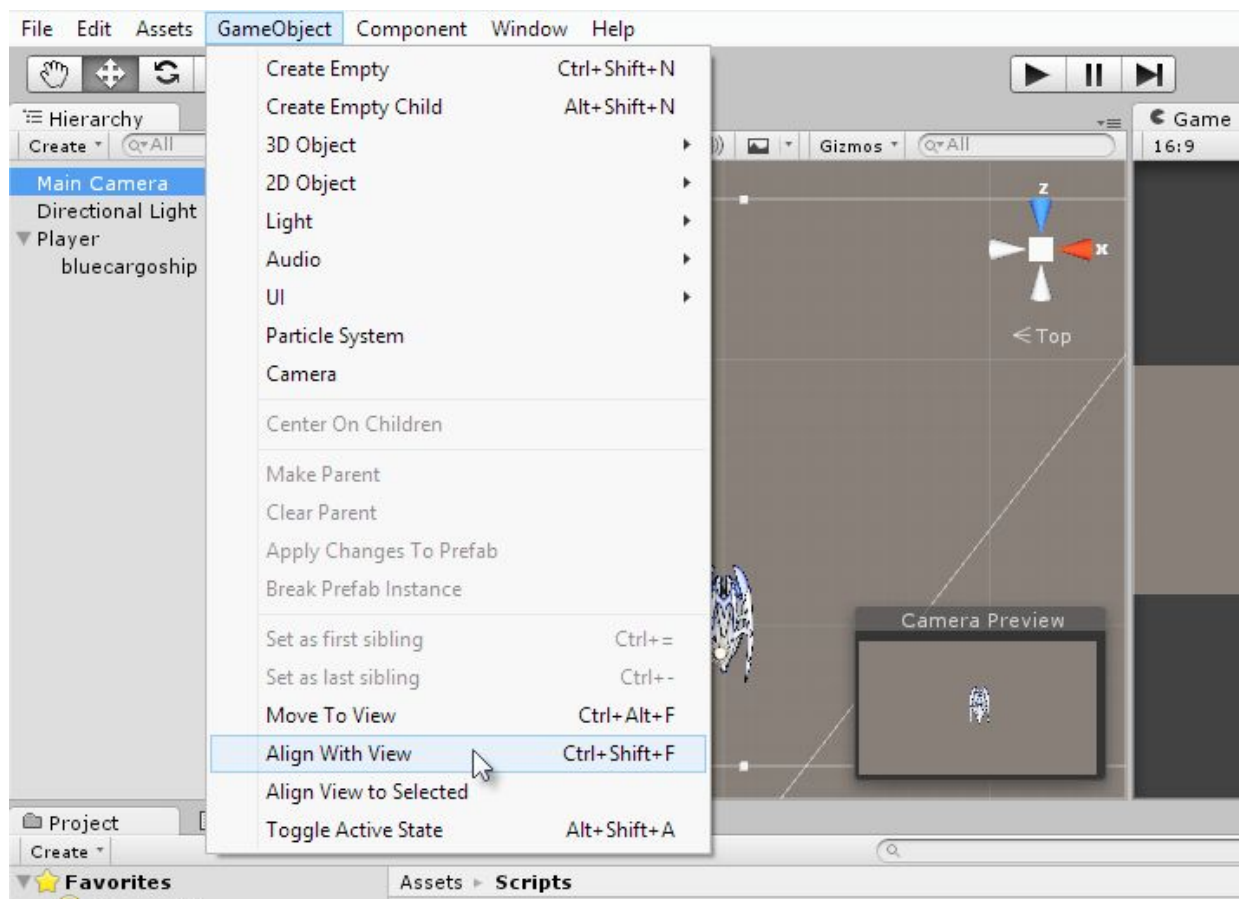


图3.19 将摄像机的视角与场景（Scene）视图的视角调整一致

经过视角的调整，游戏看起来已经好多了，但是这里仍然存在一个问题：当游戏在进行过程中时，宇宙飞船并没有按照预期始终面向着光标。这是因为当前的摄像机仍然是一个透视（**Perspective**）类型的，从而导致了屏幕坐标和世界坐标之间进行转换时产生了意想不到的结果。为了修正这个错误，需要将摄像机的类型转化为正交

（**Orthographic**）类型，这个类型的摄像机是一个专门的2D类型的摄像机，因此没有那种透视的纵深感。首先在场景中选中摄像机，然后在检查（**Inspector**）面板中，将**Projection**的值由透视（**Perspective**）设定为正交（**Orthographic**），如图3.20所示。

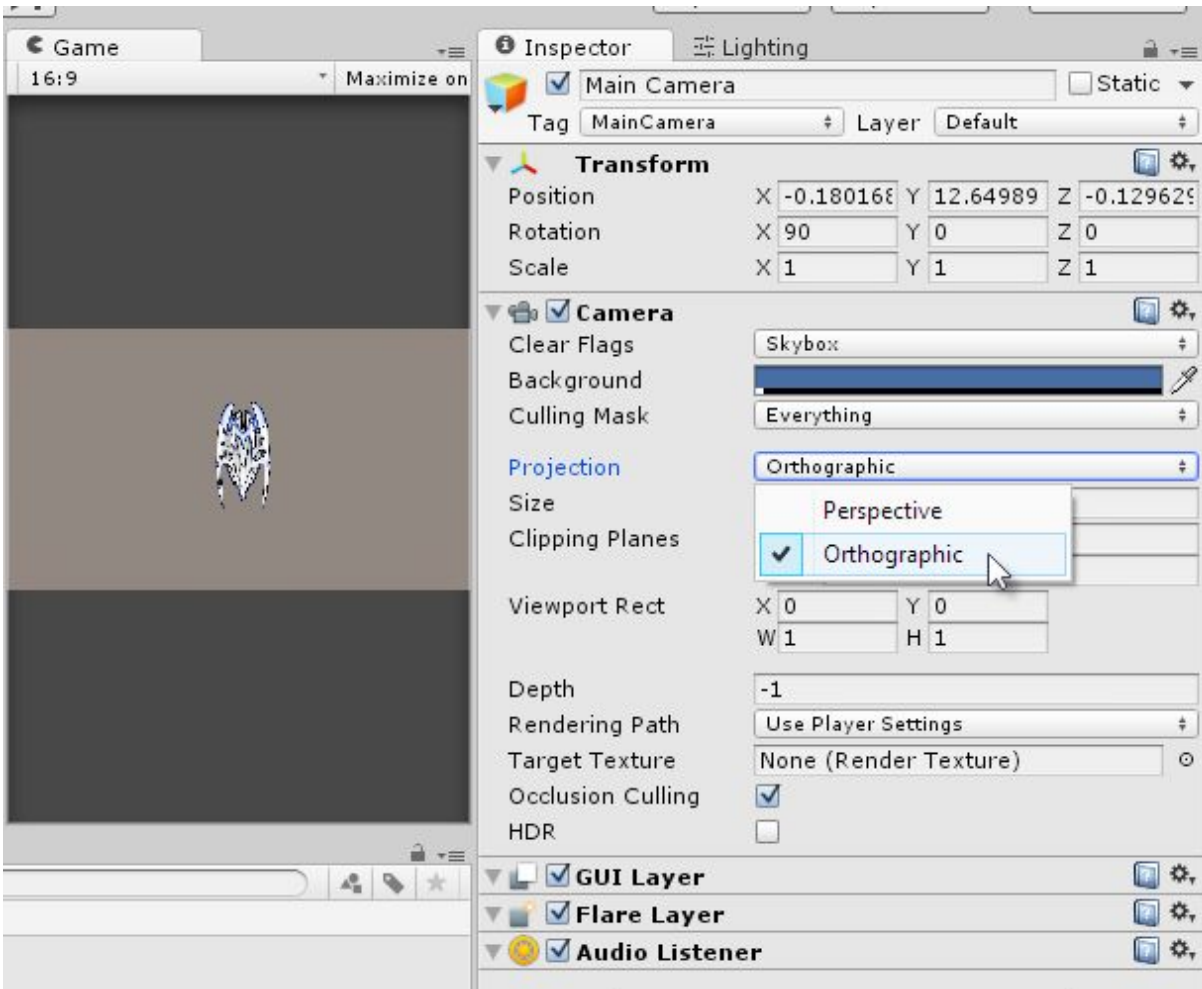


图3.20 将摄像机的类型改为正交（Orthographic）

所有正交类型的摄像机在对象检查（Inspector）面板中都有一个Size属性，这个属性是透视（Perspective）摄像机所不具有的，它决定了游戏世界中的单元对应着游戏屏幕像素的数量。游戏世界的单元与游戏屏幕像素最好是一一对应的关系，这样贴图就会以正常的尺寸显示，同时光标也能按照预期的效果运动。游戏设计的分辨率应该是全高清（Full HD）的，也就是1920×1080，屏幕高宽比为16:9。对于这个分辨率，正交类型摄像机的Size属性值应该为5.4，如图3.21所示。选择这个值的原因超出了本书的讨论范围，但是这里可以给出一个公式，

即屏幕的高（单位为像素）/ 2 / 100。因此，这里就是 $1080 / 2 / 100 = 5.4$ 。

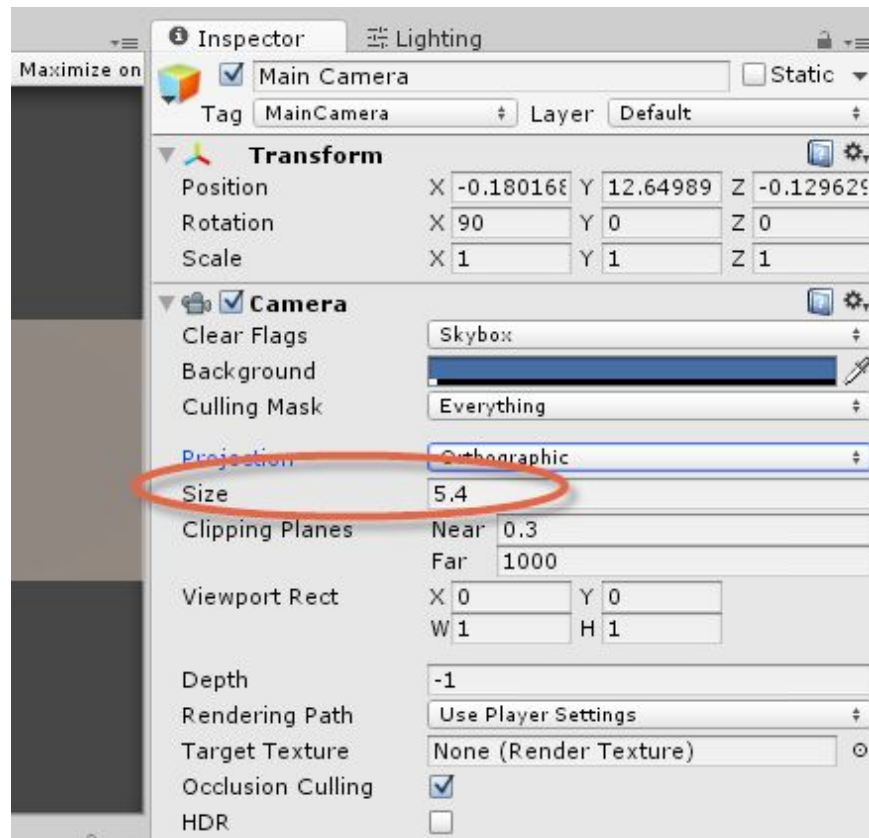


图3.21 将摄像机的Size属性设定为5.4以保证1:1的转换率

最后，确认游戏（Game）选项卡的视图已经按照16:9的比例来展示游戏，如果当前比例不正确，可以在游戏（Game）选项卡进行调整，单击游戏（Game）选项卡上方左侧的角，然后在弹出的下拉列表框中选中16:9，如图3.22所示。



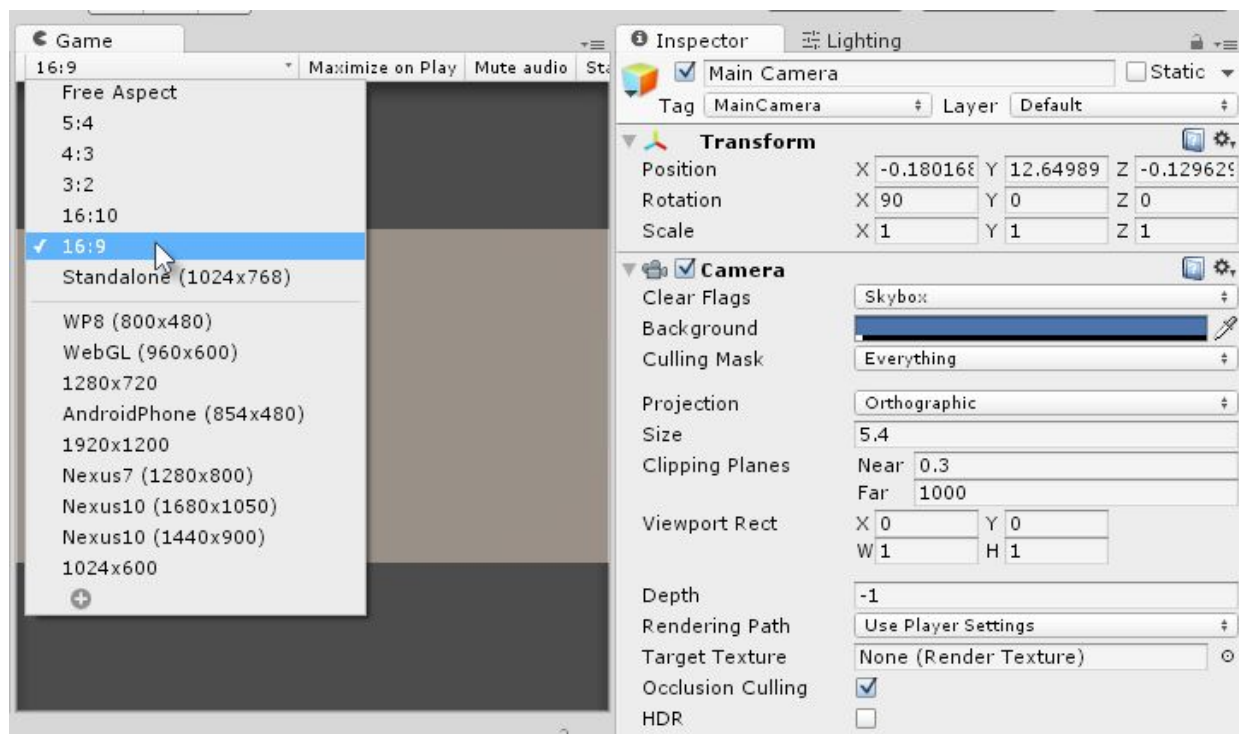


图3.22 将游戏按照16:9的比率进行演示

现在来测试这个游戏的运行，此时已经拥有了一个可以通过按下“W”“A”“S”“D”来控制运动，通过移动光标来控制转向的宇宙飞船对象，如图3.23所示，这个游戏已经初具雏形了，不过仍然有很多工作要做。

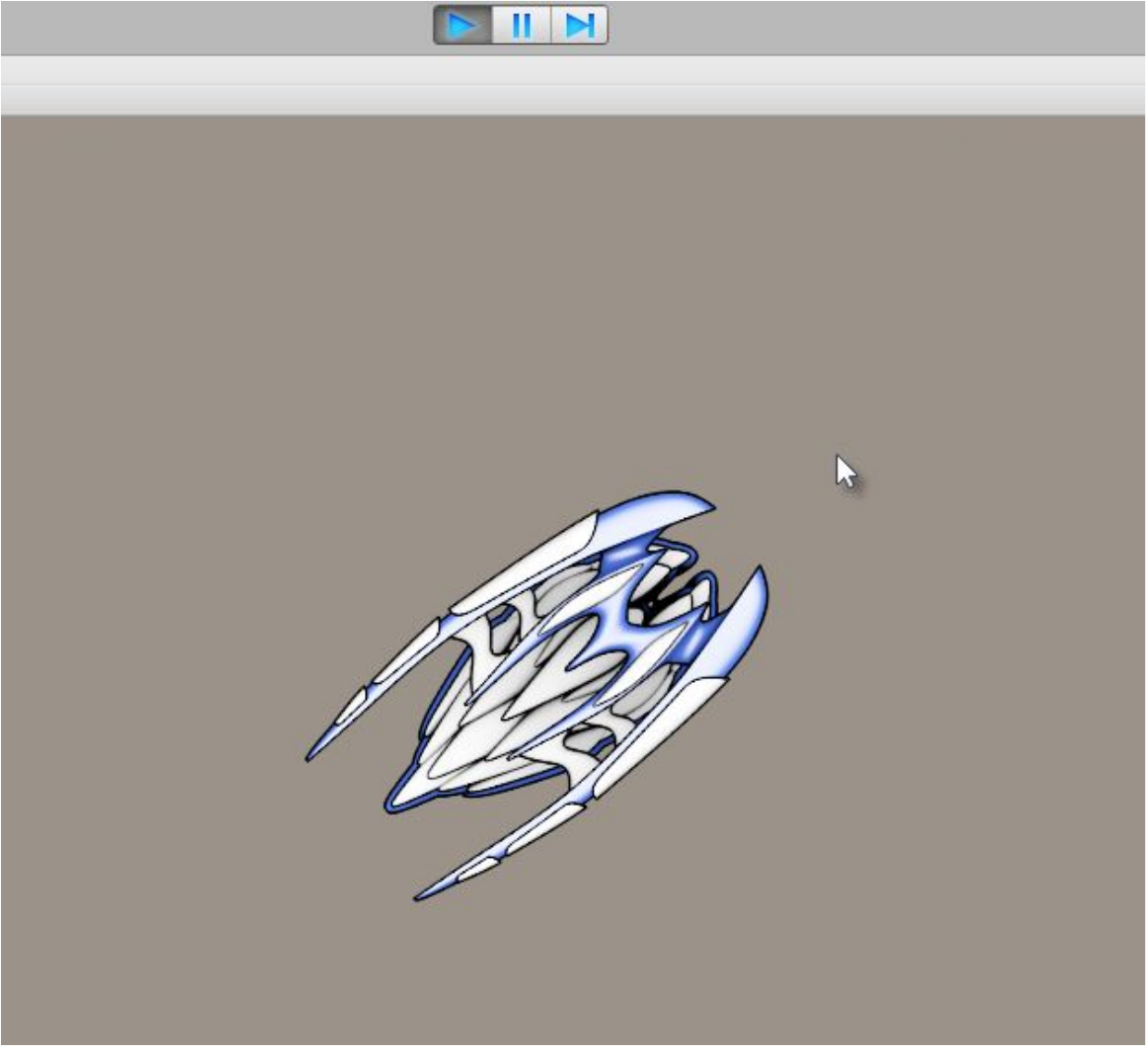


图3.23 调整船体来面向光标

## 3.6 范围的锁定

现阶段的宇宙飞船看起来有些过于庞大了。不过这一点很容易修改，只需要调整玩家（Player）的“Scale”属性。这里将“Scale”属性里面的X、Y、Z的值都设置为0.5，如图3.24所示。不过不管如何修改宇宙飞船的尺寸，这里都存在着一个问题：当玩家控制着飞船时，飞船可以

自由地飞到屏幕的边界之外。这就意味着在玩游戏时，飞船很有可能飞出边界，然后在游戏中消失，而且再也不出现。实际上我们希望的是相机一直保持静止的同时，**Player**对象的移动一直在摄像机的视野中，而且它永远都不会超出这个视野。

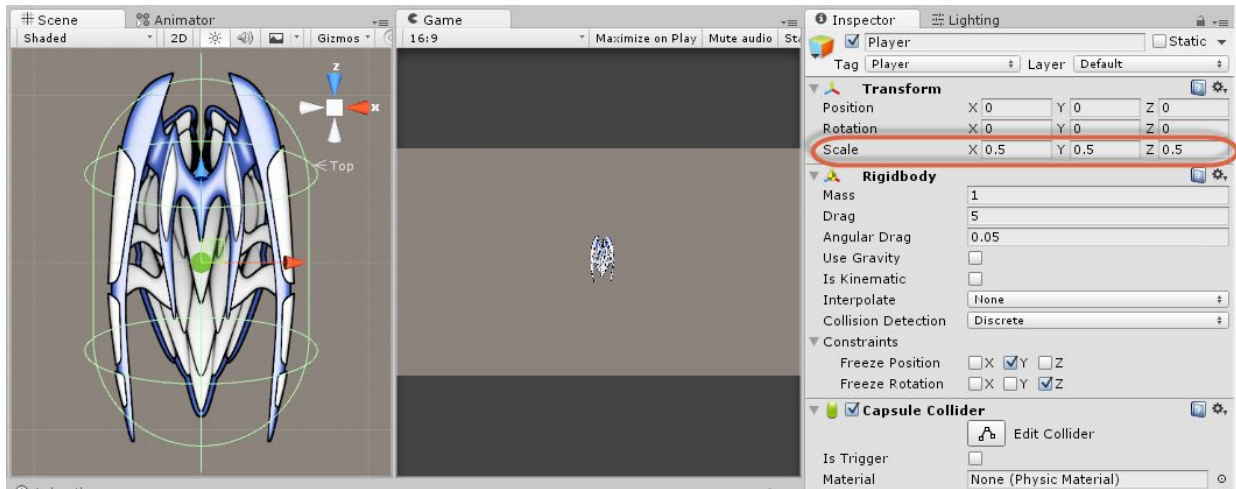


图3.24 重新定义玩家（Player）的大小

这里有很多种方法可以实现游戏对象活动范围的锁定，使用最多的是通过编码来实现。其中一种方法就是将玩家（Player）的位置值限定在一个特定的范围内，在最大值和最小值之间。下面的代码示例3.2是一个名为“BoundsLock”的C#类，这个脚本应该依附在玩家（Player）上。

### 代码示例3.2:

```
//-----  
using UnityEngine;  
using System.Collections;  
//-----  
public class BoundsLock : MonoBehaviour  
{  
    //-----  
    private Transform ThisTransform = null;
```

```

public Vector2 HorzRange = Vector2.zero;
public Vector2 VertRange = Vector2.zero;
//-----
// 初始化函数
void Awake ()
{
    ThisTransform = GetComponent();
}
//-----
// 在每一帧都会调用Update
void LateUpdate ()
{
    //限制它的位置
    ThisTransform.position = new Vector3(Mathf.Clamp
        (ThisTransform.position.x, HorzRange.x, HorzRange.y),
        ThisTransform.position.y,
        Mathf.Clamp(ThisTransform.position.z, VertRange.x,
            VertRange.y));
}
//-----
}
//-----

```

下面对代码示例3.2进行总结。

- **LateUpdate()**是在所有的**FixedUpdate()**和**Update()**函数调用之后执行的，这个函数允许一个对象在被渲染到屏幕之前修改它的位置。  
关于**LateUpdate**函数的更多信息可以访问  
<http://docs.Unity3d.com/ScriptReference/MonoBehaviour.LateUpdate.html>。
- 函数**Mathf.Clamp()**确保一个值的范围位于给定区间之间，这个区间由给定的最大值和最小值确定。
- 在使用**BoundsLock**脚本之前，需要将它拖动到**Player**对象上，然后指定最大值和最小值的确切数值，如图3.25所示。这些值是由游戏世界的坐标决定的，可以将游戏物体移动到摄像机所能观察的边缘，然后根据当前物体的**Transform**组件来确定这两个值。

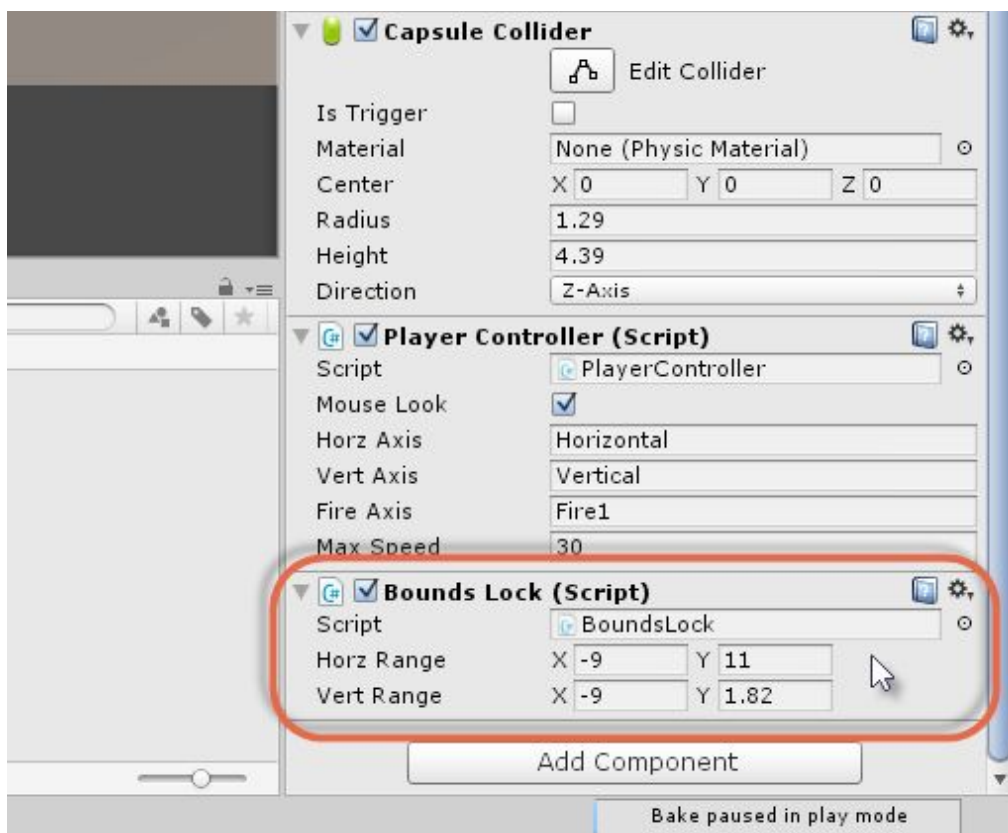


图3.25 对“Bounds Lock”进行设定

单击工具栏上的“Play”按钮对游戏进行测试。现在玩家宇宙飞船始终都在视野中飞行，而不会超出屏幕的边界。

## 3.7 生命值

玩家宇宙飞船和敌人都需要生命值。生命值是一个用来判断场景中角色是否存在的标准，通常这个值的区间为0~100。0意味着死亡，而100意味着最健康的状态。虽然生命值对于每一个实例都是独立的（玩家宇宙飞船有自己的生命值，而每个敌人都有自己独立的生命值），但是所有实例的生命值都有很多共同之处，因此可以专门来编写一个生命值的类，然后将所有需要生命值的对象都附加一个这样的

类。代码示例3.3就是这样的一个类，它应该附加到玩家和所有的敌人或者其他需要生命值的对象上。

### 代码示例3.3:

```
using UnityEngine;
using System.Collections;
//-----
public class Health : MonoBehaviour
{
    public GameObject DeathParticlesPrefab = null;
    private Transform ThisTransform = null;
    public bool ShouldDestroyOnDeath = true;
    //-----
    void Start()
    {
        ThisTransform = GetComponent();
    }
    //-----
    public float HealthPoints
    {
        get
        {
            return _HealthPoints;
        }

        set
        {
            _HealthPoints = value;

            if(_HealthPoints <= 0)
            {
                SendMessage("Die",
                    SendMessageOptions.DontRequireReceiver);

                if(DeathParticlesPrefab != null)
                    Instantiate(DeathParticlesPrefab,
                        ThisTransform.position, ThisTransform.rotation);

                if(ShouldDestroyOnDeath)
                    Destroy(gameObject);
            }
        }
    }
    //-----
    [SerializeField]
    private float _HealthPoints = 100f;
```



```
}  
//-----
```

下面对代码示例3.3进行总结。

- **Health**类中通过“\_HealthPoints”私有变量来控制对象的生命值，这个变量可以通过“HealthPoints”属性来访问，该属性具有一个Get访问器和一个Set访问器。Get访问器用来读取“HealthPoints”的值，Set访问器用来设置“HealthPoints”的值。
- “\_HealthPoints”变量被声明为“SerializeField”（序列化字段），这样就可以在Inspector面板中看到这个值。在运行和调试代码时，可以随时地观察到玩家的当前生命值。
- **Health**类是一个典型的事件驱动编程例子，是因为这个类会在Update()函数中不断地去检查对象生命值的状态。当对象生命值下降到0时，就意味着对象死亡。对死亡的检查是在C#属性中的Set方法中进行的，这样做的原因在于Set方法中是唯一的health值不会发生变化的地方。这就意味着Unity可以在每一帧都节省大量的工作。
- **Health**类中使用了SendMessage()函数，这个函数允许通过名字来调用依附在对象上的任何组件所提供的公共函数。在这个游戏中，调用的函数就是一个名为“Die”的函数（如果对象的组件上提供了这个函数）。如果对象的组件上不存在名字匹配的函数，那么就不会执行任何函数。这是一种轻松快捷地调用对象上自定义行为的方法。这种方法是与类型无关的。但是，SendMessage()函数的缺点也很明显，它的内部会调用一个名为Reflection()的过程，这个过程十分耗费资源，而且执行缓慢。基于这个原因，除了触发死

亡事件或者类似事件之外，很少调用SendMessage()函数，因为这些事件都不是频繁发生的。更多关于SendMessage()函数的信息可以在Unity的在线文档<http://docs.Unity3d.com/ScriptReference/GameObject.SendMessage.html>处找到。

- 当生命值降到0以下时，会触发死亡条件，脚本会触发一个代表死亡的粒子系统，这时会出现一个对象死亡的效果。

当Health脚本附加到玩家宇宙飞船上之后，它在检查（Inspector）面板以一个组件的形式出现。这个组件包含一个名为“Death Particles Prefab”的属性，该属性是可选的（可以是空的），当对象死亡的时候，播放一个粒子系统的效果。有了这个属性之后，在设置对象死亡时的爆炸或者鲜血四溅的效果时就容易了很多，如图3.26所示。

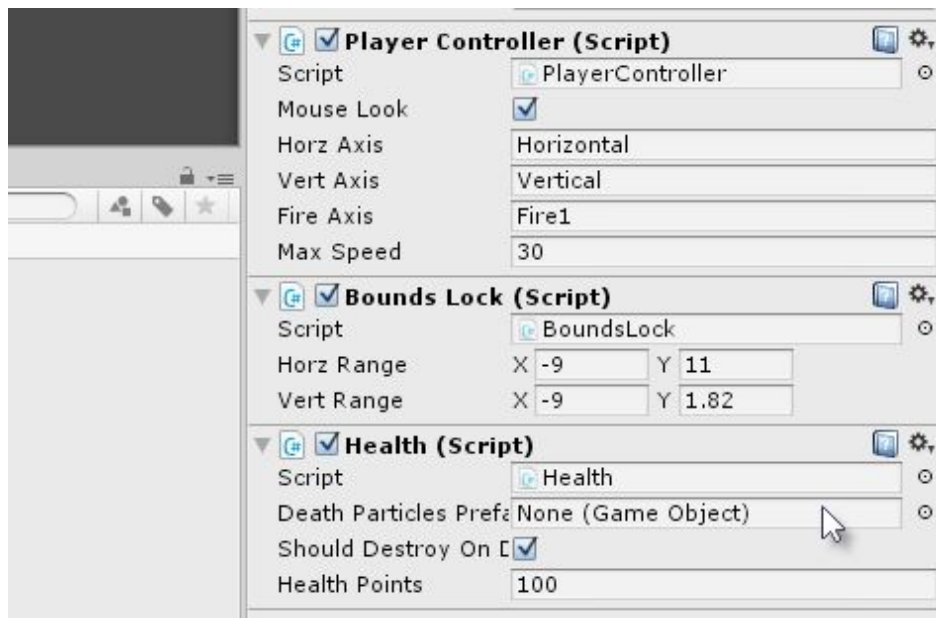


图3.26 附加的生命值（Health）组件

## 3.8 死亡和粒子系统

在这个双轴游戏中，玩家和敌人都是宇宙飞船。当它们受到伤害时，就会爆炸成为一个火球，只有这种效果才足够真实。为了实现这种爆炸效果，可以使用粒子系统。粒子系统指的是一种特殊的对象，它有两个主要部分组成，管道（或者发射器）和粒子。发射器（Emitter）是指在游戏世界里产生或者生成新粒子的部分，粒子（Particles）指的是产生之后沿着特有的轨迹运行的小物体或者小碎片，总之，粒子系统是在游戏中制造雨、雪、雾、闪光、爆炸等效果的最佳选择。可以使用菜单选项中的“GameObject | Particle System”来从头开始创造自己的粒子系统，或者使用一些系统Unity中预置的粒子系统。在这个游戏中，要使用一些预置的粒子系统（Particle System）。首先，在应用程序菜单中选中“Assets | Import Package | ParticleSystems”，如图3.27所示。

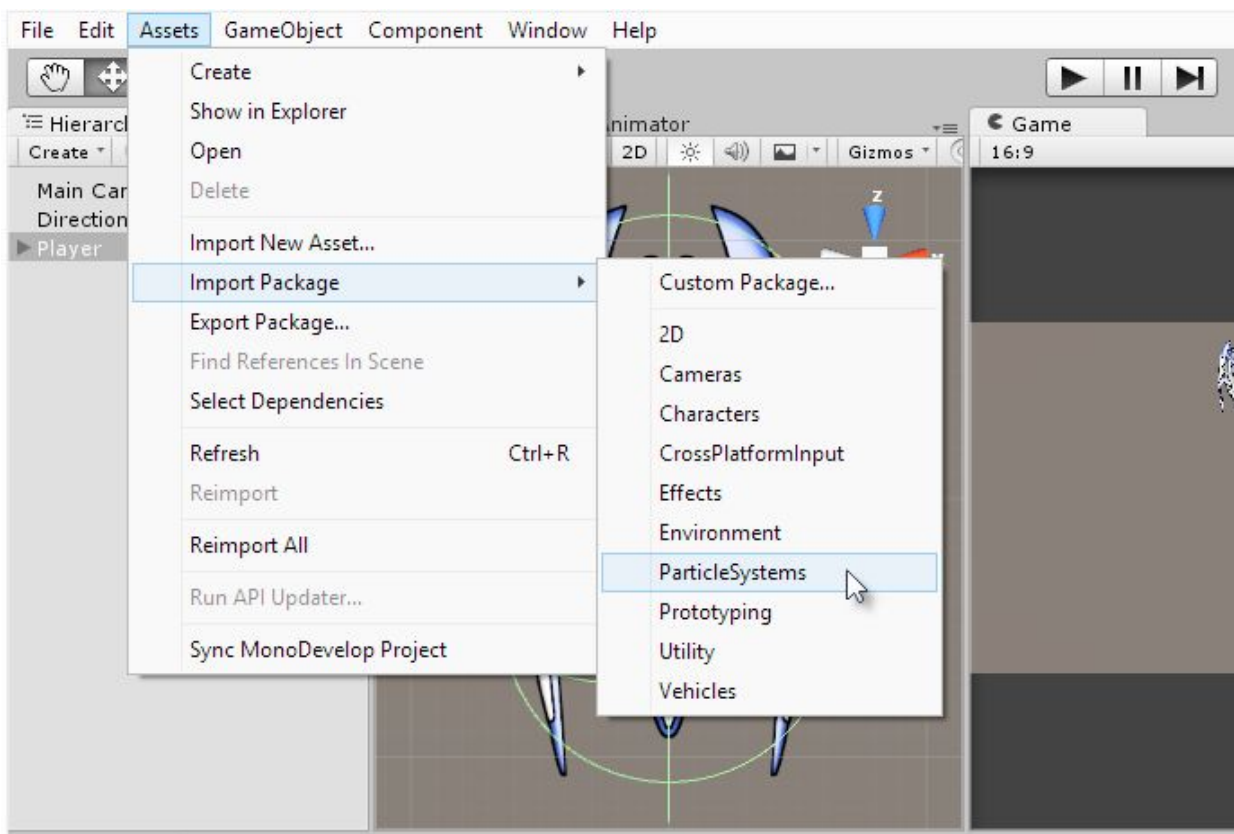


图3.27 向项目中导入一个粒子系统（Particle System）

当Import对话框出现之后，保持所有原有的设置不变，然后单击“Import”按钮，导入包括所有粒子系统在内的完整资源包。现在在项目（Project）面板里的“Standard Assets | ParticleSystems | Prefabs”文件夹中就增加了一个“ParticleSystems”文件夹，如图3.28所示。可以通过将所有的预设体拖曳到场景中来测试这些粒子系统的效果。注意，只有在场景中选中一个粒子系统，才可以看到它的预览效果。

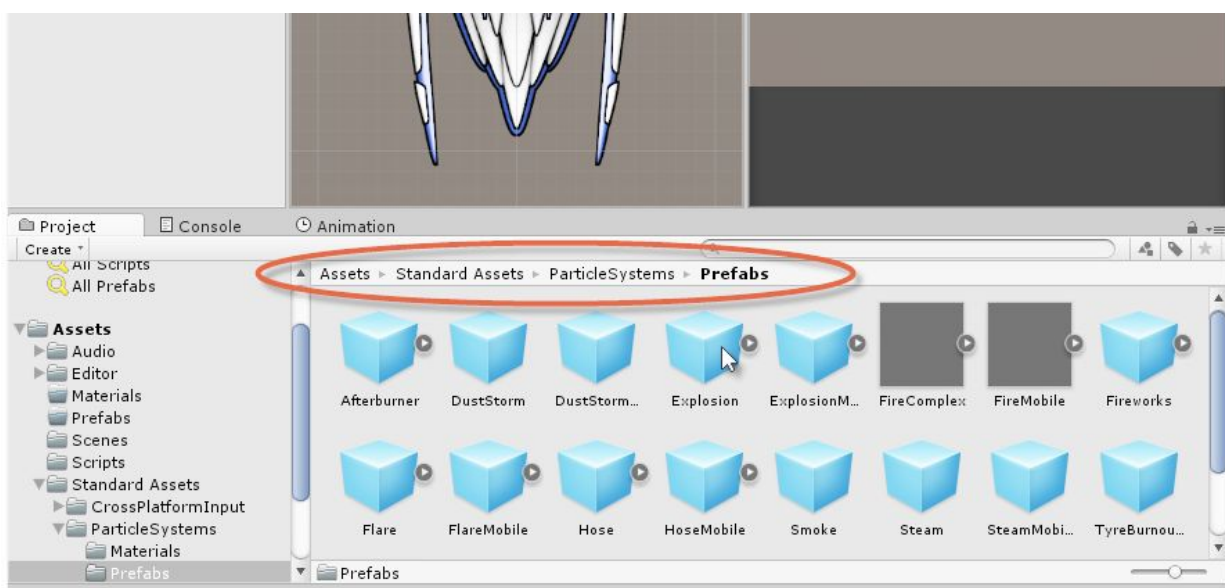


图3.28 向项目（Project）面板里导入粒子系统（Particle System）

如图3.28所示，默认的资源包中包含了一个爆炸（Explosion）系统，这绝对是一个好消息。为了测试这个爆炸系统，可以将爆炸（Explosion）系统拖曳到场景中，然后按下工具栏上的“Play”键。现在已经完成了大部分工作，不过还差一点。现在已经看到了一个十分合适的粒子效果，然后可以将这个系统拖曳到对象检查（Inspector）面板中的Health组件的“Death Particles Prefab”属性中。这个粒子系统的工作

流程就是：当有玩家或者敌人死亡时，将会触发爆炸系统，创造出一个爆炸效果。但是这个粒子系统永远都不会被销毁。这是一个缺陷，因为每当场景中的一个敌人死亡，就会产生一个新的粒子系统。而当大量的敌人死亡后，现场将充满了废弃的粒子系统，这样对系统的性能和存储来说都是不利的。设想一下满场景都是无用的粒子系统，将会给系统带来多大的负担。因此必须修正这个爆炸系统，创造一个新的适合游戏的预设体。为了实现这个功能，需要先将当前的爆炸系统拖曳到场景中，如图3.29所示。

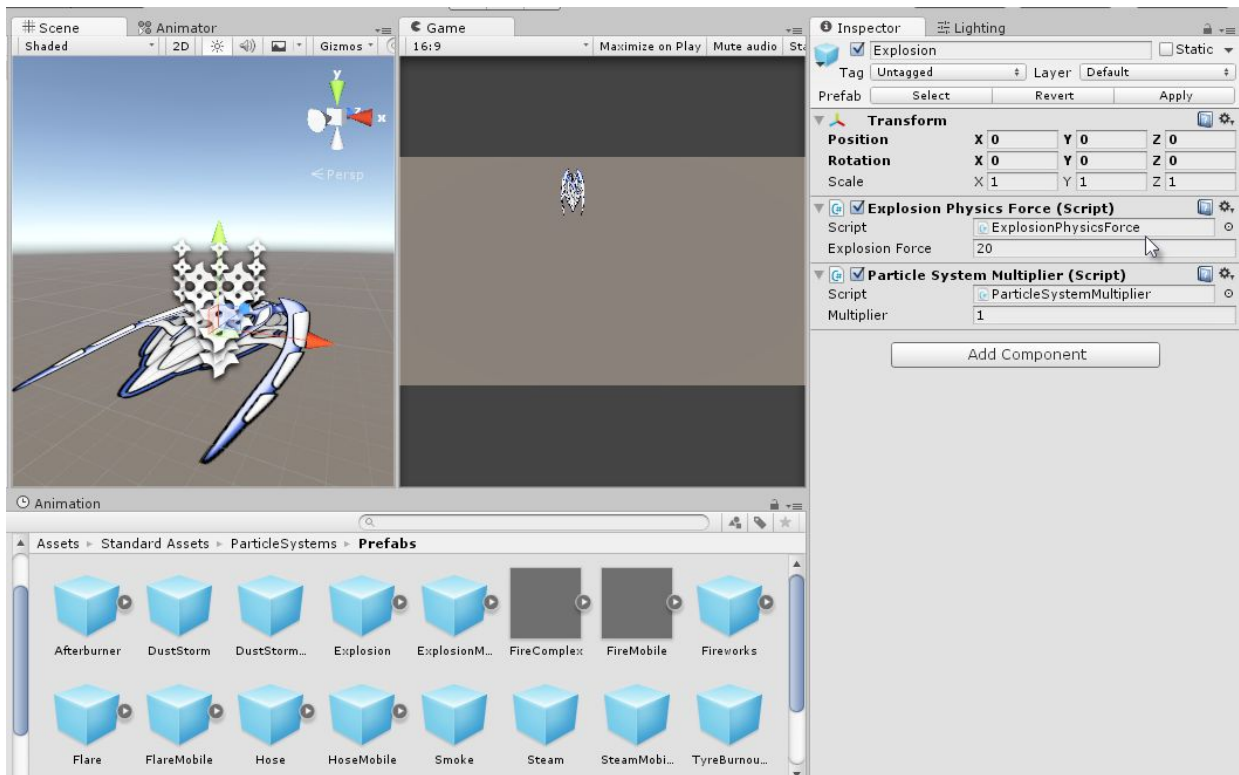


图3.29 将要修改的爆炸系统拖曳到场景中

接下来，必须重新对粒子系统（Particle System）进行定义，这样粒子系统（Particle System）在实例化过后很快就会被销毁。经过这样的安排，每一个产生的爆炸效果最终都会销毁。为了使一个对象能够

在一定时间后自我销毁，需要创建一个新的C#脚本，把这个脚本命名为TimeDestroy.cs，代码示例3.4中给出了详细的内容。

### 代码示例3.4:

```
//-----  
using UnityEngine;  
using System.Collections;  
//-----  
public class TimedDestroy : MonoBehaviour  
{  
    public float DestroyTime = 2f;  
    //-----  
    //初始化函数  
    void Start ()  
    {  
        Invoke("Die", DestroyTime);  
    }  
  
    void Die ()  
    {  
        Destroy(gameObject);  
    }  
    //-----  
}  
//-----
```

下面对代码示例3.4进行几点总结。

- **TimedDestroy**类会在经过一定的时间间隔（**DestroyTime**）后销毁它所附加的对象。
- **Invoke()**函数会在**Start**事件中调用，它会在特定的时间间隔结束后执行一次指定名称的函数，注意仅仅执行一次，时间以秒为单位。
- 跟**SendMessage()**函数一样，**Invoke()**函数也需要依赖于**Reflection**。基于这个原因，为了保证系统的性能，应该谨慎使用该函数。



- 在经过指定时间的间隔之后，**Die()**函数将会被**Invoke()**函数所调用，以此来销毁游戏对象（例如一个粒子系统）。

现在将**TimedDestroy**脚本拖动到场景中爆炸粒子系统上，然后按下工具栏上的“**Play**”键完成对代码的测试，经过一定的时间间隔之后，对象就会被销毁，在对象检查（**Inspector**）面板中可以修改这个时间间隔的值，如图3.30所示。

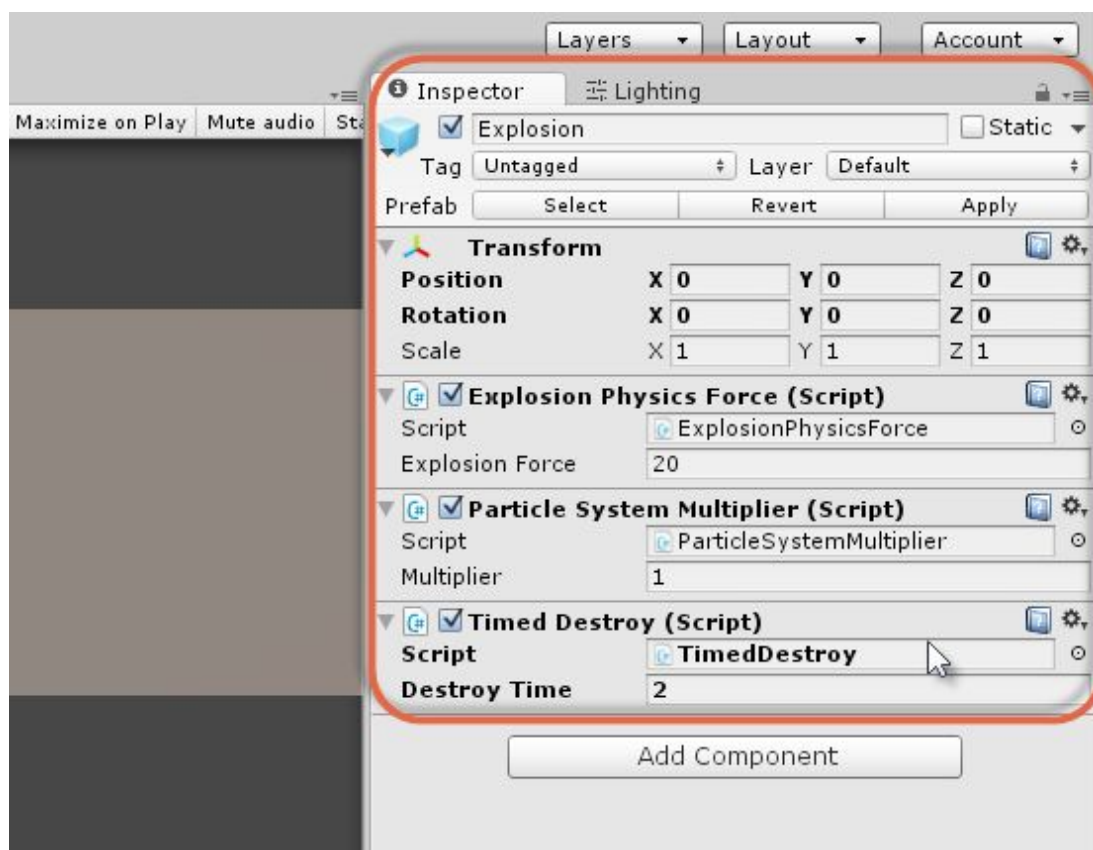


图3.30 向一个爆炸粒子系统添加**TimedDestroy**脚本

**TimedDestroy**脚本应该在时间到期时移除爆炸粒子系统。现在需创建一个新的、完全独立的预设体，为了实现这一点，首先在层次（**Hierarchy**）面板上将爆炸系统重命名为**ExplosionDestroy**，然后将这

个系统从层次（Hierarchy）面板上拖曳到项目（Project）面板上的 Prefabs 文件夹中。Unity 会自动地创建一个新的预设体，来表示修改以后的粒子系统，如图3.31所示。

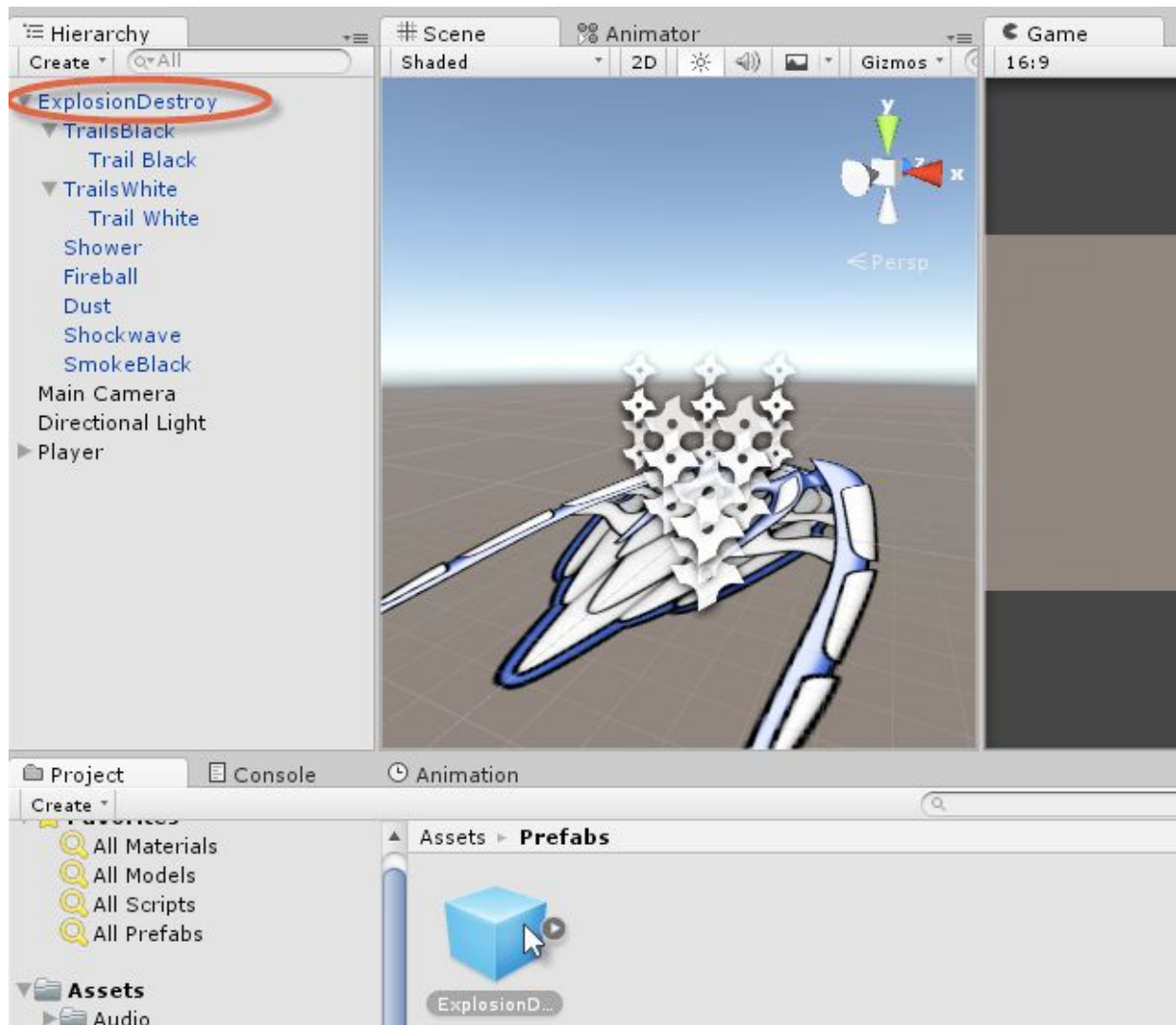


图3.31 创建一个爆炸预设体

现在，将新创建的预设体拖动到玩家（Player）的对象检查（Inspector）面板中Health 组件的“Death Particle System”位置。这样做

确保了当玩家（Player）对象死亡的时候，爆炸预设体将被实例化，如图3.32所示。

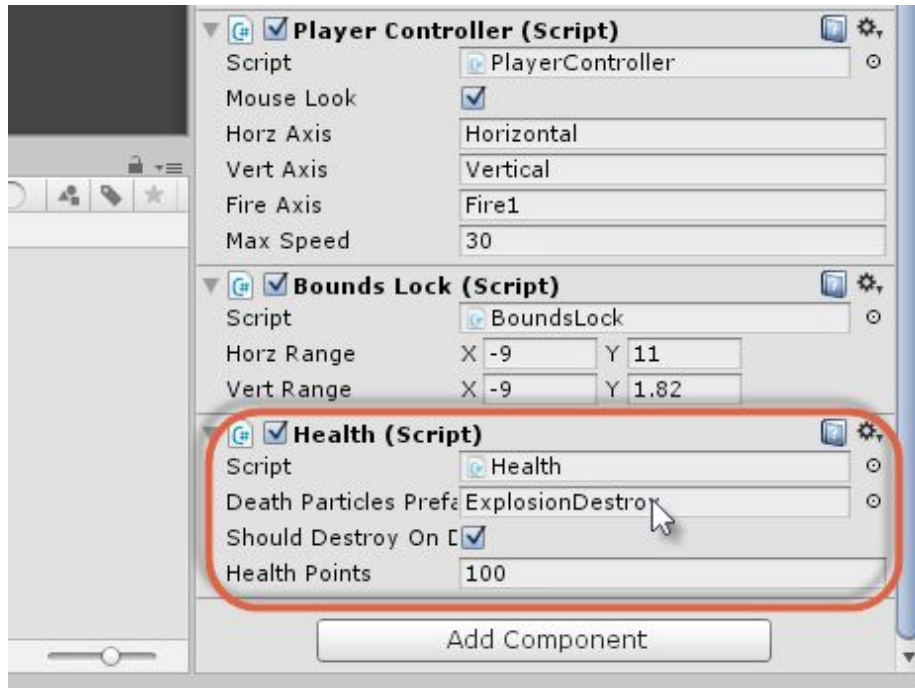


图3.32 对Health脚本进行配置

现在开始运行这个游戏，会发现无法启动玩家的死亡事件，因此无法测试粒子系统是否能正常产生。在这个场景中不存在任何能对玩家造成伤害的敌人，同样也无法从检查（Inspector）面板将玩家的Health值手动设置为0。可以在Health脚本中添加一些用来测试死亡的功能，当空格键按下时，就会触发死亡事件。代码示例3.5中给出了修改之后的Health脚本。

### 代码示例3.5:

```
//-----  
using UnityEngine;  
using System.Collections;  
//-----
```

```

public class Health : MonoBehaviour
{
    public GameObject DeathParticlesPrefab = null;
    private Transform ThisTransform = null;
    public bool ShouldDestroyOnDeath = true;
    //-----
    void Start()
    {
        ThisTransform = GetComponent();
    }
    //-----
    public float HealthPoints
    {
        get
        {
            return _HealthPoints;
        }

        set
        {
            _HealthPoints = value;

            if(_HealthPoints <= 0)
            {
                SendMessage("Die",
                    SendMessageOptions.DontRequireReceiver);

                if(DeathParticlesPrefab != null)
                    Instantiate(DeathParticlesPrefab,
                        ThisTransform.position, ThisTransform.rotation);

                if(ShouldDestroyOnDeath)Destroy(gameObject);
            }
        }
    }
    //-----
    void Update()
    {
        if(Input.GetKeyDown(KeyCode.Space))
            HealthPoints = 0;
    }
    //-----
    [SerializeField]
    private float _HealthPoints = 100f;
}
//-----

```

再运行一下游戏，注意，此时游戏中的**Health**脚本已经被修改过了，可以随时在键盘上按下空格键来触发玩家的死亡事件。当按下了空格键以后，玩家（**Player**）对象就会被销毁，然后粒子系统就会产生，当指定时间耗尽之后，粒子系统也会销毁。现在已经有有了一个十分好玩的可控制对象了，这个对象也有了生命值和死亡的功能，如图3.33所示，一切看起来都不错。

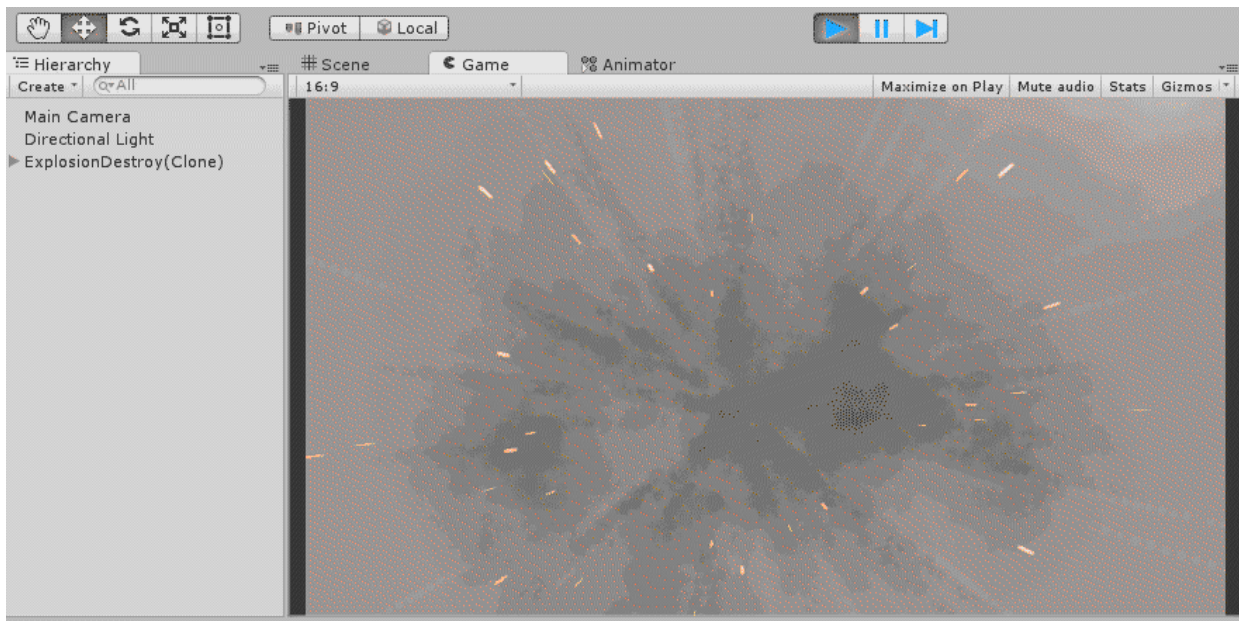


图3.33 触发爆炸粒子系统

## 3.9 敌人

接下来的步骤就是要为玩家创建可以用来射击和摧毁的目标了，但是这些目标同时也可能会摧毁玩家，也就是说这些目标就是游戏中的敌人。这些敌人具有太空船的外形，散布在整个场景中，而且每隔一定的时间间隔就会产生一些新的敌人。这些敌人会在场景中跟随着玩家（**Player**）对象，而且逐渐逼近。从本质上说，每个敌人都应该是

多种复杂行为协同工作的产物，因此，这些复杂行为都应该由各自独立的脚本实现，下面对它们逐个进行介绍。

- **Health:** 所有的敌人都支持生命值的功能，每一个敌人在场景出现时都具有一定的生命值，当它们的生命值下降到0以下时，就会被销毁，之前已经为这个功能创建了一个**Health**脚本。
- **Movement:** 每一个敌人都会处在不断的运动中，按照运动的轨迹向前直行。也就是说，每一个敌人都会持续地沿着它的方向前进。
- **Turning:** 每一个敌人都会转动，从而保持一直面对着**Player**对象，即使**Player**对象一直在运动。将**Turning** 和**Movement**功能相结合，就可以保证敌人永远都朝着**Player**对象运动。
- **Scoring:** 每当一个敌人销毁的时候，就会给予玩家一定的奖励，也就是说，敌人的死亡会增加玩家的得分。
- **Damage:** 每一个敌人都会通过碰撞对**Player**对象造成伤害，所有的敌人不能射击，但是当它们靠近**Player**对象时，就对其造成伤害。

现在已经明确了敌人应该具有的功能，接下来就在场景中创建一个敌人。首先制作一个具体的敌人，然后利用这个敌人再创建一个预设体，利用这个预设体就可以实例化出来很多敌人。首先选中场景中的玩家（**Player**）角色，然后按下“**Ctrl+D**”组合键，或者从应用程序菜单上依次选中“**Edit | Duplicate**”，这样就可以创建出第二个玩家（**Player**）对象，如图3.34所示。



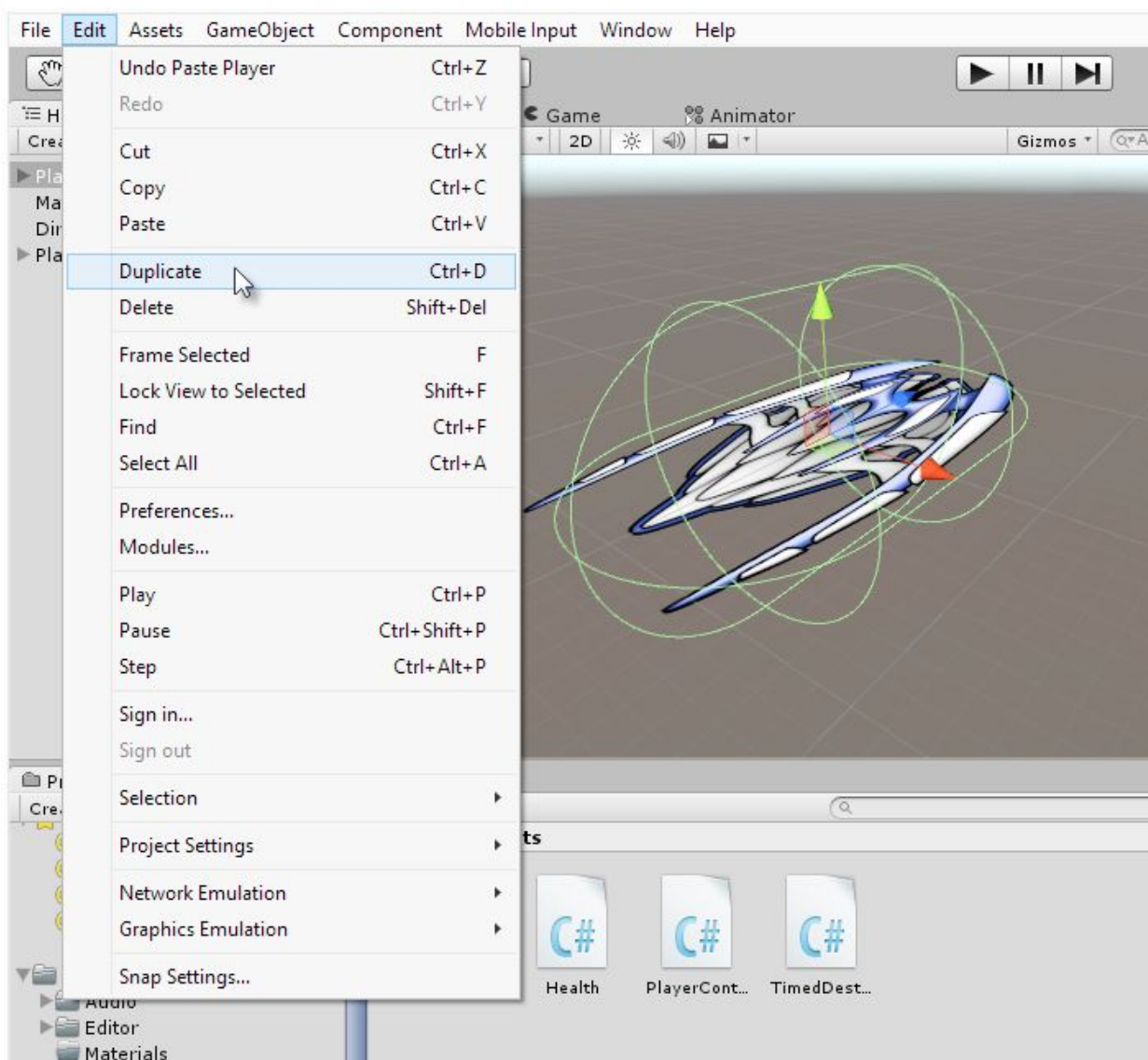


图3.34 对Player对象进行复制

将这个对象命名为“Enemy”，并确保对象的Tag属性值不是“Player”，在整个场景中只能有一个对象的Tag名字是“Player”，当然这个对象就是真正的“Player”对象。另外，需要暂时停止所有Player对象的功能，以便能将所有的注意力都放在敌人（Enemy）对象上，如图3.35所示。

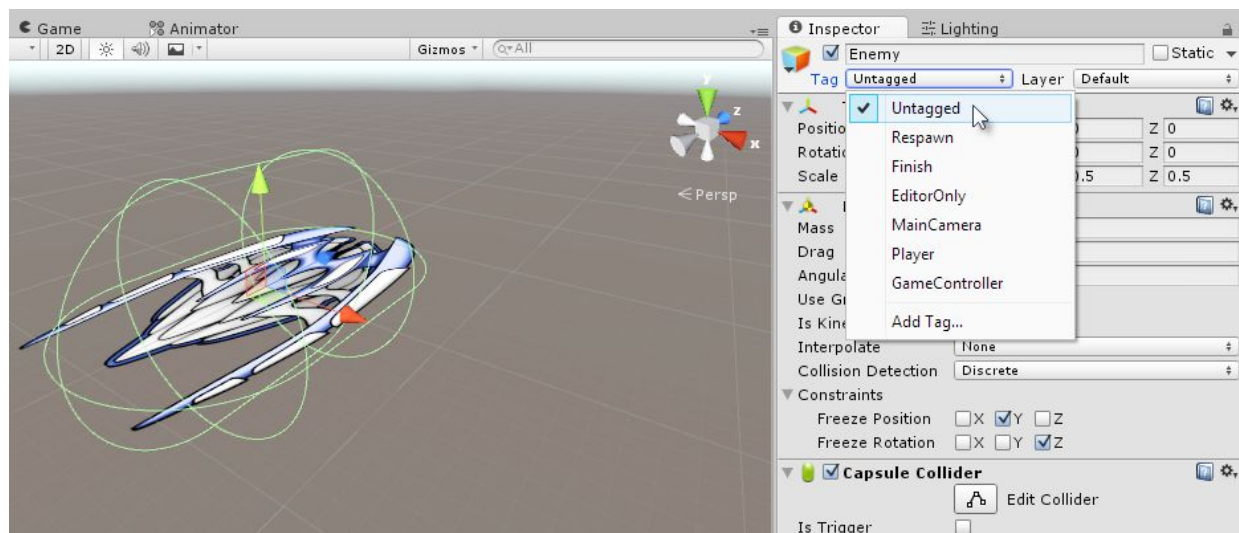


图3.35 将敌人（Enemy）对象上的Tag属性去除Player

首先选中复制出来的敌人（Enemy）对象的Sprite子对象，在对象检查（Inspector）面板中选中“Sprite Renderer”组件，为其选中一个新的图片精灵对象。为敌人（Enemy）角色选择一个黑颜色的飞船，这个图片精灵就会在视图对对象进行更新，如图3.36所示。

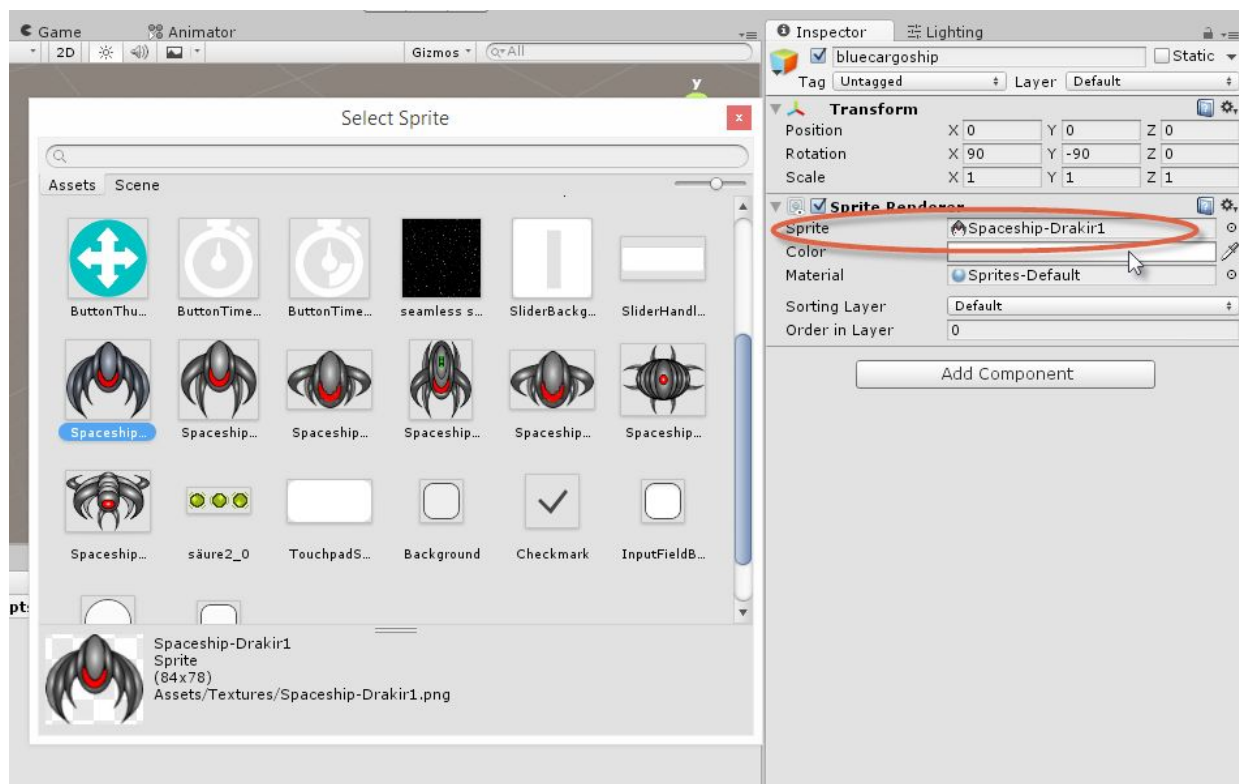


图3.36 为“Sprite Renderer”组件选择一个图片精灵对象

在将图片精灵转换成一个敌人（**Enemy**）角色之后，需要修改“**Rotation**”值，来保证图片精灵与父对象对齐，尤其是要注意图片精灵的朝向要与父级的前向向量相同，如图3.37所示。



图3.37 修改敌人（Enemy）图片精灵的Rotation属性

选中敌人（Enemy）的父级对象，然后移除“Rigidbody”“PlayerController”以及“BoundsLock”等组件，记住，“Health”组件要保留下来，如图3.38所示，因为在游戏中，敌人也需要一个生命值。另外，要对胶囊碰撞体组件的体积进行调整，以便大小更好地适合敌人（Enemy）对象。

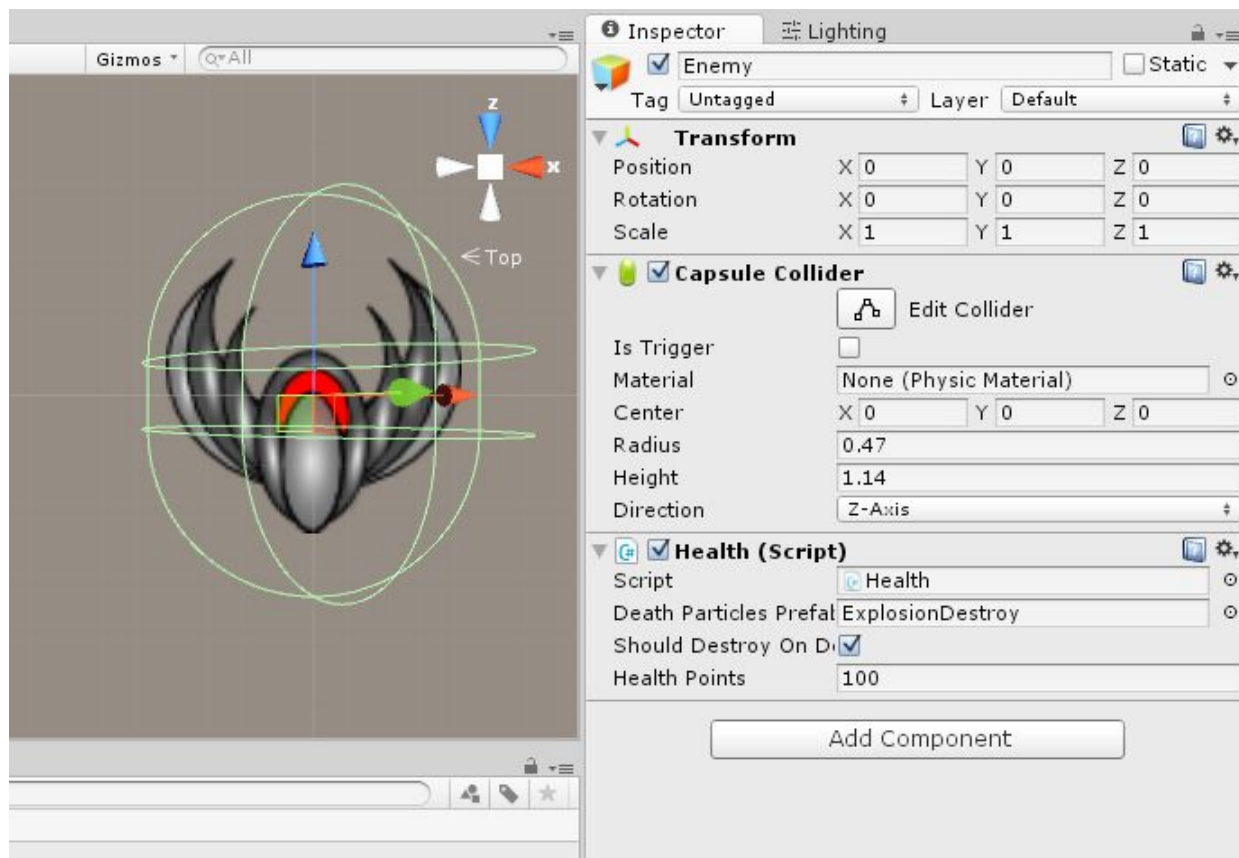


图3.38 调节“Enemy”图片精灵的Rotation值

接下来，开始为敌人（**Enemy**）对象编码，重点是敌人（**Enemy**）对象的运动。敌人（**Enemy**）对象应该以一个指定的速度不断地向前运动。为了实现这个功能，需创建一个名为“**mover.cs**”的脚本文件。这个脚本文件将会被附加到敌人（**Enemy**）对象上。下面的代码示例3.6给出了这个功能的实现代码。

### 代码示例3.6:

```
//-----
using UnityEngine;
using System.Collections;
//-----
public class Mover : MonoBehaviour
{
```

```

//-----
private Transform ThisTransform = null;
public float MaxSpeed = 10f;
//-----
// 初始化部分
void Awake ()
{
    ThisTransform = GetComponent<Transform>();
}
//-----
// 每帧调用一次Update函数
void Update ()
{
    ThisTransform.position += ThisTransform.forward * MaxSpeed *
        Time.deltaTime;
}
//-----
}
//-----

```

对代码示例3.6做以下总结。

- **Mover**脚本中实现了对象按照一个特定的速度（每秒运动 **MaxSpeed**）朝着它的前向向量的方向运动，为了实现这个功能，需要使用**Transform**组件。
- **Update**函数负责更新对象的位置。简而言之，它使用前向向量 **Forward**与对象速度的乘积再加上当前的位置就可以得出对象的下一个位置。**Time.deltaTime**值是用来产生一个与游戏帧速率无关的效果，这样运动的单位就是每秒，而不是每帧。关于**deltaTime**更详细的信息可以访问Unity的在线文档<http://docs.Unity3d.com/ScriptReference/Time-deltaTime.html>来获得。

经常对代码进行测试是一个很好的习惯。单击工具栏上的“**Play**”键运行代码，游戏中的敌人（**Enemy**）可能运动太慢或者太快。如果出现了这种情况，就停止测试返回到游戏设计中来。选中场景中的敌人



（Enemy），从对象检查（Inspector）面板处的Mover组件中修改Max Speed的值，如图3.39所示。

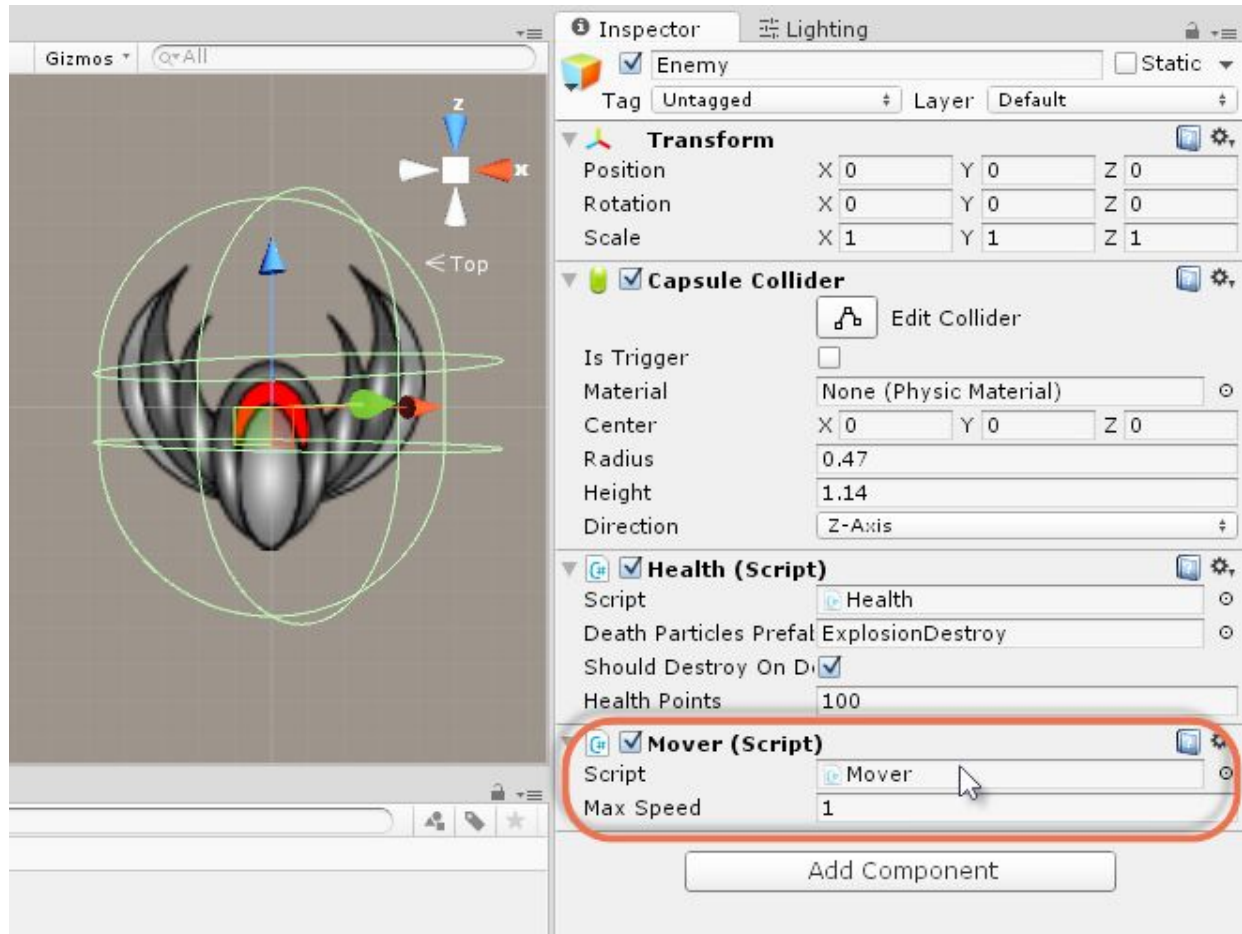


图3.39 修改敌人（Enemy）对象的速度

除了沿着直线运动之外，“Enemy”对象还应该一直都朝着玩家（Player）对象运动，为了实现这个功能，需要编写另外一个和“Player Controller”脚本功能相类似的脚本文件。当玩家（Player）对象转动面向光标的时候，敌人（Enemy）对象也应该转动方向去面向玩家（Player）对象，该功能通过代码实现，将这部分代码保存为一个名

为“ObjFace.cs”的新脚本文件。这个脚本将会被附加到敌人（Enemy）对象上，如代码示例3.7所示。

### 代码示例3.7:

```
//-----
using UnityEngine;
using System.Collections;
//-----
public class ObjFace : MonoBehaviour
{
    //-----
    public Transform ObjToFollow = null;
    public bool FollowPlayer = false;
    private Transform ThisTransform = null;
    //-----
    // 初始化函数
    void Awake ()
    {
        //获得当前组件的Transform属性
        ThisTransform = GetComponent();

        //应该面对Player对象
        if(!FollowPlayer)return;

        //获得Player对象的Transform属性
        GameObject PlayerObj =
            GameObject.FindGameobjectWithTag("Player");
        if(PlayerObj != null)
            ObjToFollow = PlayerObj.GetComponent();
    }
    //-----
    // Update函数应该在每一帧调用一次
    void Update ()
    {
        //跟随目标对象
        if(ObjToFollow==null)return;

        //获取跟随对象的方向
        Vector3 DirToObject = ObjToFollow.position -
            ThisTransform.position;

        if(DirToObject != Vector3.zero)
            ThisTransform.localRotation = Quaternion.LookRotation
                (DirToObject.normalized,Vector3.up);
    }
    //-----
}
```

```
}  
//-----
```

对代码示例3.7进行如下分析和总结。

- **ObjFace**函数应该经常转动一个对象，这样这个对象的前进向量才能一直指向场景中的目标。
- 在**Awake**事件中，调用了**FindGameObjectWithTag**函数通过**Tag**名来进行检索对象，在整个场景中只有一个对象的**Tag**名为“**Player**”，这个对象应该就是玩家（**Player**）宇宙飞船对象。玩家（**Player**）对象的位置就是“**Enemy**”对象的目标。
- 系统会在每一帧都自动调用**Update()**函数，然后产生一个对象当前位置与目的位置的位移向量，这个向量表示这个对象应该行进的方向。函数**Quaternion.Look Rotation()**会接收这个向量，然后按照它调整对象前向向量的方向。这样就保证了对象始终面向着目的地，关于**LookRotation()**函数的详细内容可以参阅Unity的在线文档<http://docs.Unity3d.com/ScriptReference/Quaternion.LookRotation.html>。

在测试这段代码之前，一定要先确认场景中的**Tag**属性设置为“**Player**”的对象已经被启用，敌人也和这个对象有一定的距离。在对象检查（**Inspector**）面板中找到**ObjFace**组件的**Follow Player**复选框并勾选上。当完成这个操作之后，敌人就会一直面向着“**Player**”对象运动了，如图3.40所示。

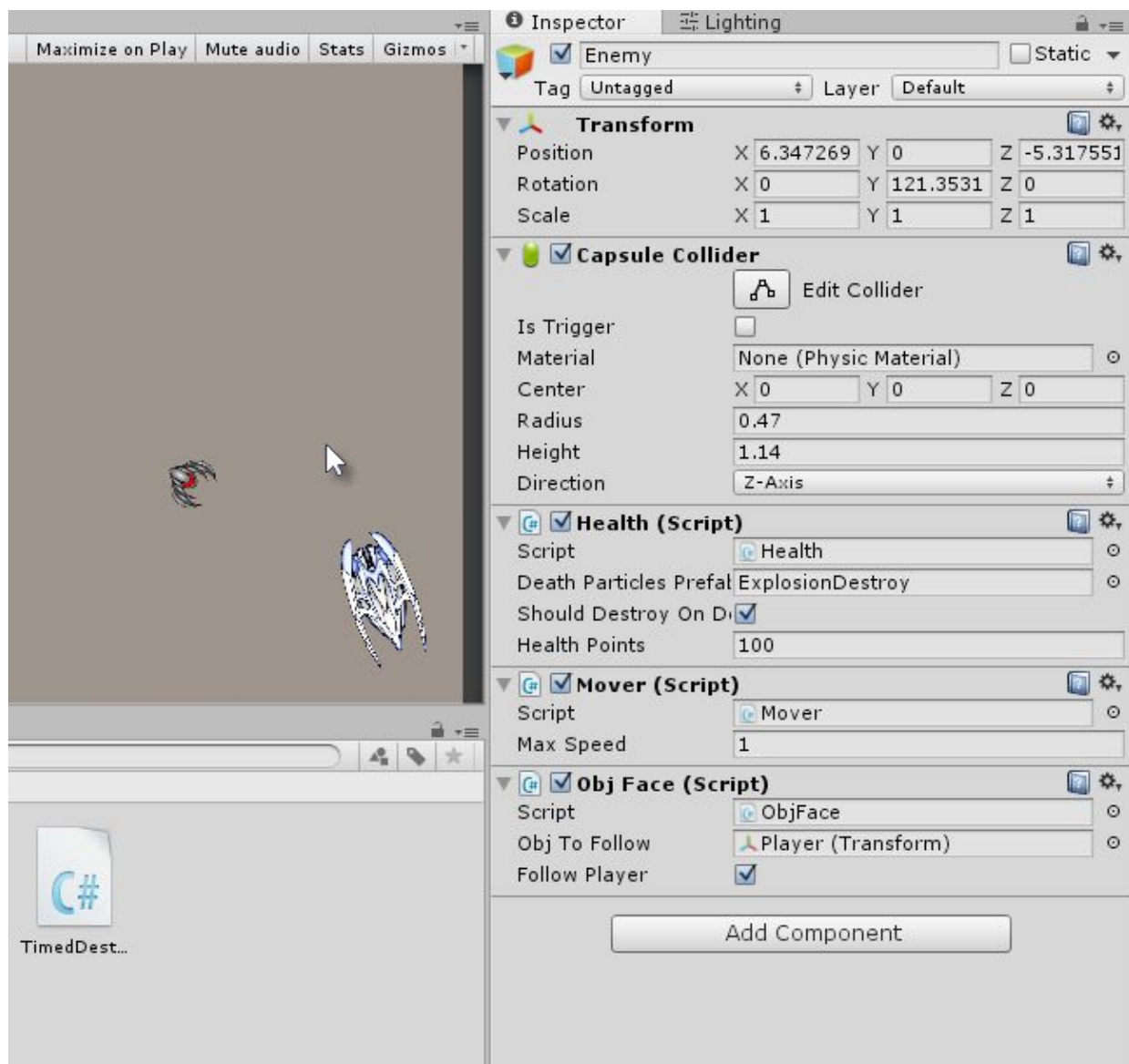


图3.40 敌人飞船一直面向着玩家（Player）对象运动

当敌人与玩家（Player）对象发生碰撞之后，就会对玩家（Player）对象造成伤害，甚至可能会杀死玩家（Player）对象。为了实现这个设计，必须可以侦测到敌人与玩家（Player）对象之间的碰撞。选中代表敌人的敌人（Enemy）对象，然后在检查（Inspector）面板中的胶囊碰撞体（Capsule Collider）组件中勾选上“Is Trigger”复选

框。通过这样的调整之后，敌人（Enemy）对象就变成了一个虚体，可以穿过玩家（Player），但是不会引起碰撞，如图3.41所示。

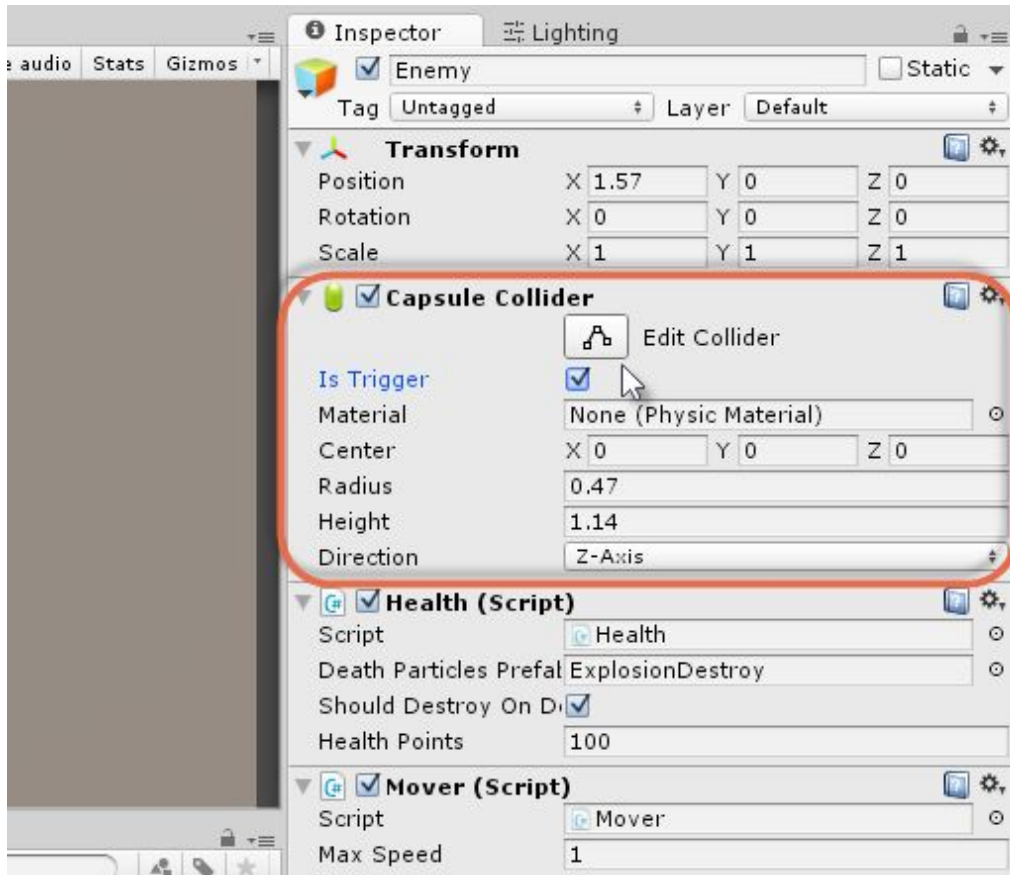


图3.41 将敌人（Enemy）对象的碰撞器改为一个触发器（Trigger）

接下来创建一个用来检测碰撞的脚本，这段脚本也能处理在碰撞发生时对玩家（Player）对象造成的伤害，代码示例3.8给出了详细内容（ProxyDamage.cs），这段代码附加到敌人（Enemy）对象上。

### 代码示例3.8:

```
//-----  
using UnityEngine;  
using System.Collections;  
//-----  
public class ProxyDamage : MonoBehaviour
```

```
{
//-----
//每秒造成的伤害
public float DamageRate = 10f;
//-----
void OnTriggerStay(Collider Col)
{
    Health H = Col.gameObject.GetComponent<Health>();

    if(H == null)return;

    H.HealthPoints -= DamageRate * Time.deltaTime;
}
//-----
}
//-----
```

下面对代码进行总结。

- 脚本ProxyDamage应该附加到敌人（Enemy）对象上，这个脚本将会通过Health组件来处理任何碰撞产生的伤害值。
- 当两个对象产生重合状态的时候，每隔一帧就会调用一次OnTriggerStay事件，在这个函数中，Health组件的HealthPoints值会以DamageRate值（这个值就是每秒的伤害）递减。

当将ProxyDamage脚本附加到“Enemy”上面之后，就可以通过在对象检查（Inspector）面板中的Proxy Damage组件来设置“Damage Rate”的值。这个值表示当碰撞发生时，“Player”对象每秒钟受到的伤害。为了加大游戏的难度，将这个值设置为100，如图3.42所示。



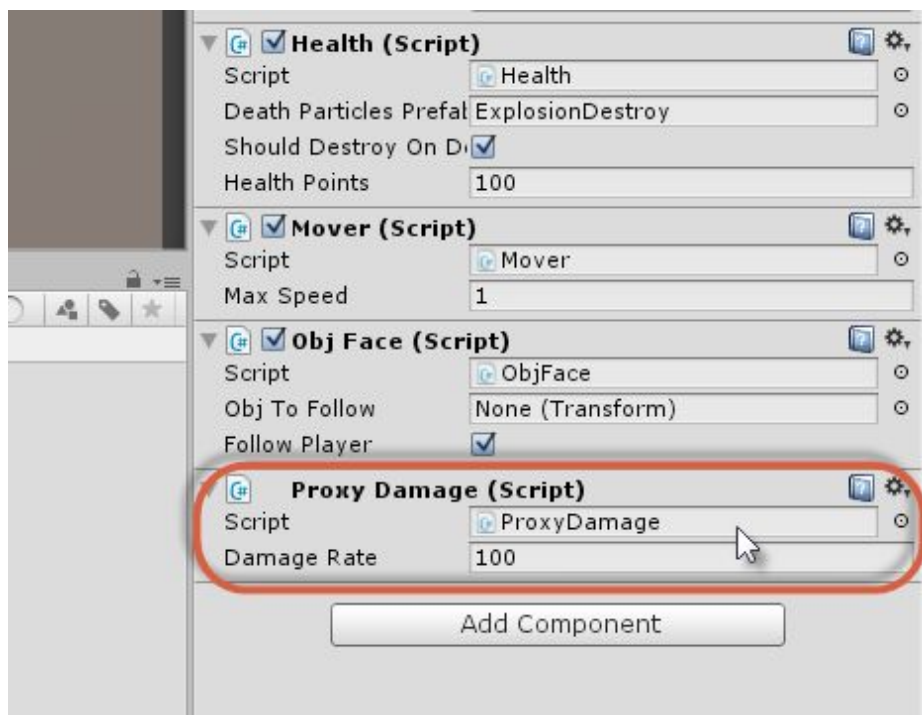


图3.42 在“Proxy Damage”组件设置“Damage Rate”的值

运行这个游戏来测试一下，按下工具栏上的“Play”键，然后试着让玩家（Player）对象和敌人（Enemy）对象之间发生一次碰撞。一秒钟之后，玩家（Player）对象就会被摧毁。一切都按照设计进行着，不过为了使游戏充满挑战性，还需要添加更多的敌人。

### 3.10 批量产生敌人

为了使关卡变得更有趣，更有挑战性，需要不止一个敌人（Enemy）。实际上，对于一个游戏来说，只要它没有结束，就应该不断地产生敌人。这些敌人是随着时间不断添加进来的。从本质来说，需要定期地产生敌人，这一节将要添加这个功能。在实现这个功能之前，需要制作敌人（Enemy）对象的prefab。操作方法为：在层次（Hierarchy）面板上选中敌人（Enemy），然后将它拖放到项目

(Project) 面板上的Prefabs 文件夹中，操作完成之后就实现了敌人 (Enemy) 预设体的制作，如图3.43所示。

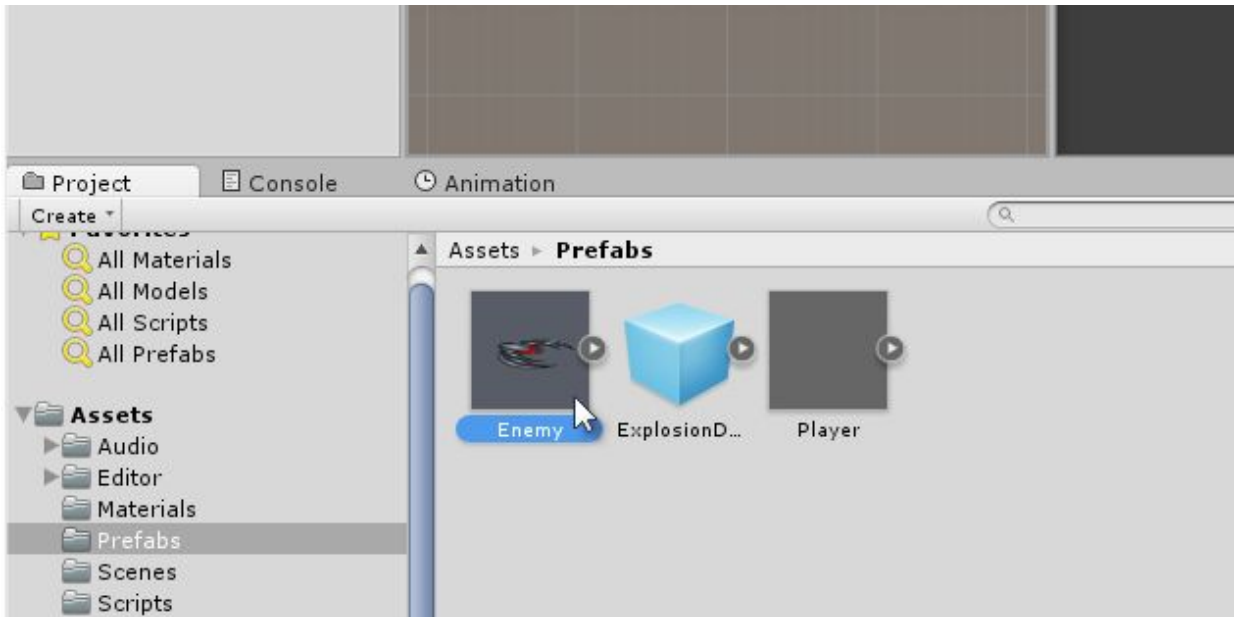


图3.43 创建一个敌人 (Enemy) 预设体

接下来创建一个新的脚本 (Spawner.cs)，这个脚本可以定期地在场景中产生指定数量的敌人 (Enemy) 对象，这个脚本应该附加到场景中一个新的空游戏对象上，如代码示例3.9所示。

### 代码示例3.9:

```
//-----  
using UnityEngine;  
using System.Collections;  
//-----  
public class Spawner : MonoBehaviour  
{  
    public float MaxRadius = 1f;  
    public float Interval = 5f;  
    public GameObject ObjToSpawn = null;  
    private Transform Origin = null;  
    //-----  
    void Awake()
```

```

{
Origin = GameObject.FindGameObjectWithTag
    ("Player").GetComponent();
}
//-----
// 初始化函数start()
void Start ()
{
    InvokeRepeating("Spawn", 0f, Interval);
}
//-----
void Spawn ()
{
    if(Origin == null)return;

    Vector3 SpawnPos = Origin.position + Random.onUnitSphere *
        MaxRadius;
    SpawnPos = new Vector3(SpawnPos.x, 0f, SpawnPos.z);
    Instantiate(ObjToSpawn, SpawnPos, Quaternion.identity);
}
//-----
}
//-----

```

下面对代码进行总结。

- **Spawner**类将会在每个**Interval**长度的时间间隔中产生一个**ObjToSpawn**的实例，**Interval**的单位是秒，产生对象的中心点为**Origin**，半径随机。
- 在**Start**事件中会调用函数**InvokeRepeating()**，通过调用这个函数就可以持续地执行**Spawn()**函数，在每个**Interval**长度的时间间隔中不断地产生新的敌人。
- 函数**Spawn()**会在场景中以原点为中心的随机位置上不断地创建**Enemy**的实例，当**Enemy**产生之后，它就会按照设计思路，一直朝着**Player**对象运动发起攻击。

**Spawner**类是一个在整个场景都支持的全局行为。这个类并不是必须依靠玩家（**Player**），同样也不是必须依靠敌人（**Enemy**）。基于这

个原因，这个类应该附加到一个空的游戏对象上。在应用程序菜单上依次选中“GameObject| Create Empty”，然后将这个对象命名为“Spawner”，最后将Spawner脚本附加到该对象上面，如图3.44所示。

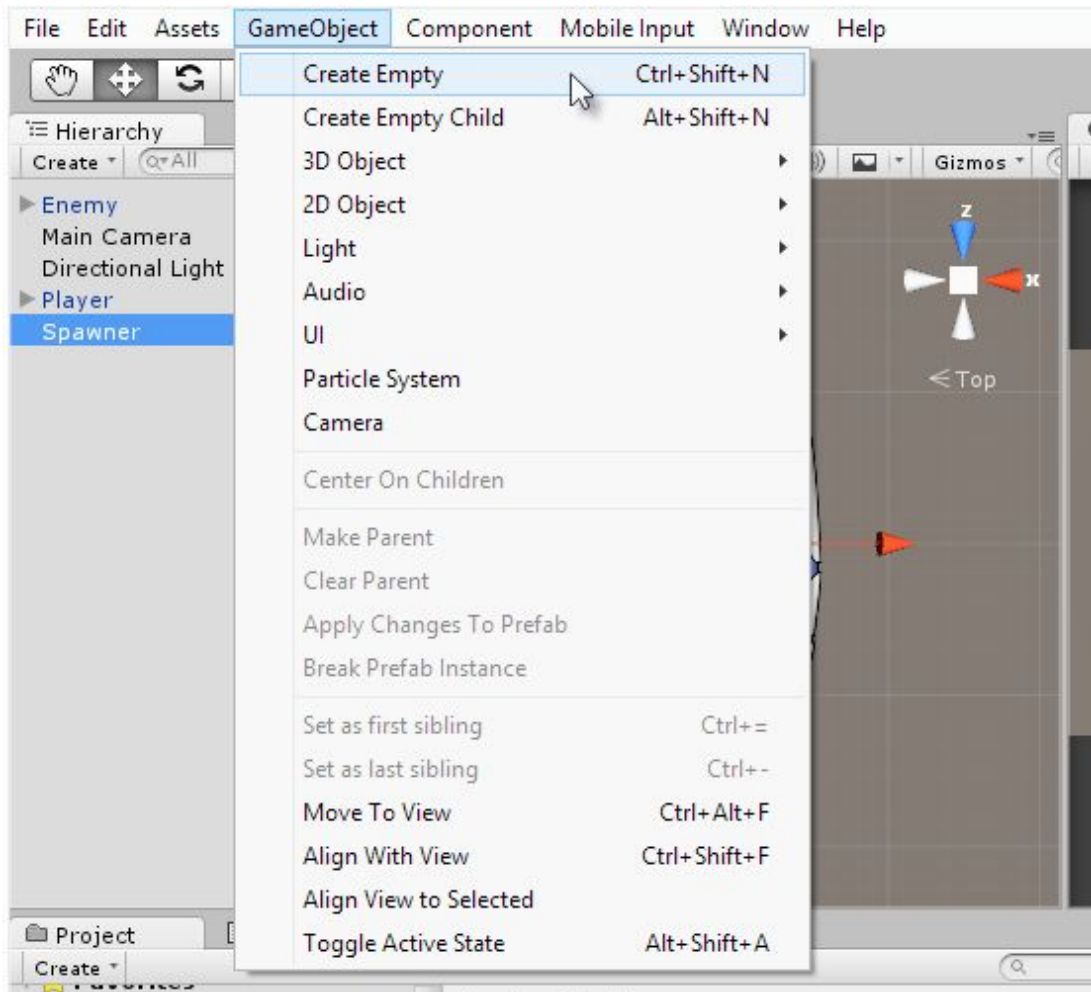


图3.44 创建一个空的游戏对象

当成功添加了对象之后，就可以在检查（Inspector）面板对其进行调整，将敌人（Enemy）预设体拖动到“Spawner”组件的“Obj To Spawn”属性中。然后如图3.45所示，将Interval的值修改为2，将“Max Radius”的值修改为5。

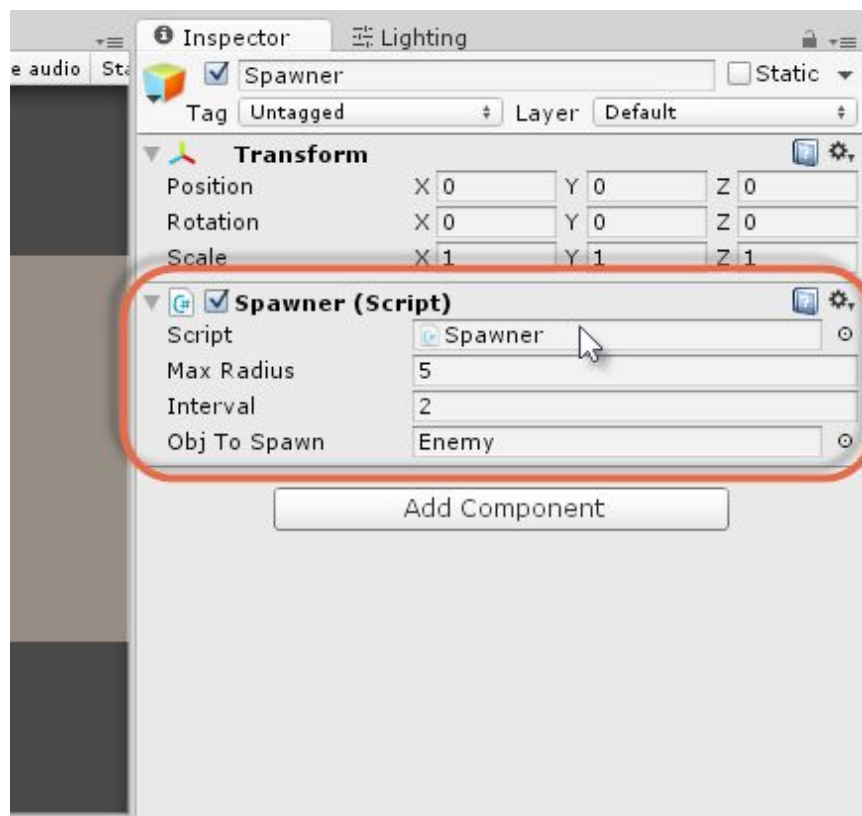


图3.45 为敌人（Enemy）对象设置Spawner组件

现在按下工具栏上的“Play”按钮来测试游戏，可以看到一个完全受玩家控制的飞船，以及无数蜂拥而来紧追飞船不放的敌人！如图3.46所示。

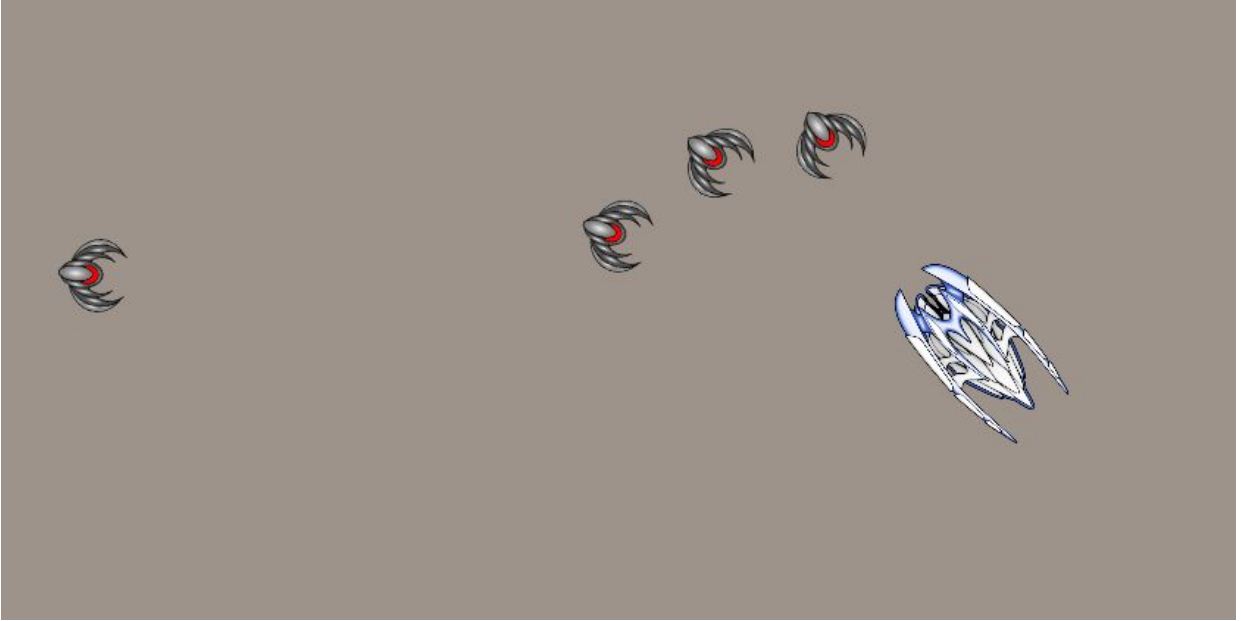


图3.46 被不断产生的敌人所包围的玩家（Player）对象

### 3.11 小结

到目前为止，太空射击游戏已经成型了：一个可以控制并遵循物理规律的玩家（Player）角色、双轴控制机制、敌人的飞船，整个场景都可能产生敌人。严格来说，所有这些加在一起，仍然不是一个真正的游戏，因为还不能开枪射击，所以不能增加得分，也不能消灭敌人。这些问题和一些其他会遇到的问题都有待改进。尽管如此，现在已经为进一步的游戏设计打下了坚实的基础，在下一章将会实现射击功能。



## 第4章 太空射击游戏 (II)

本章是上一章双轴太空射击游戏开发的继续。到现在为止，已经拥有了一个可以工作的游戏。目前游戏者可以使用运动和旋转来控制宇宙飞船。键盘上的“W”“A”“S”“D”4个键可以控制宇宙飞船的运动（上下左右），鼠标光标可以控制宇宙飞船的旋转——宇宙飞船会始终面对着鼠标光标。除了玩家控制之外，在关卡中每隔特定的周期都会产生敌人，这些敌人会在玩家的周围产生，并且会对玩家造成威胁。最后，玩家和敌人都拥有一个生命值组件，这意味着无论玩家还是敌人都可能受到伤害，甚至被摧毁。不过现在的玩家还缺乏两个最为重要的功能：发射武器，增加得分。这一章将会完善这两个以及其他的功能。如何实现武器射击功能将是一个非常有趣的问题。总体而言，本章将会包含如下主题：

- 游戏中武器和弹药的生成
- 游戏的内存管理与共享
- 游戏的声音与音频
- 游戏的计分
- 游戏的调试与测试
- 游戏的构建与发布



本章所需要的项目文件可以在本书配套文件中的“Chapter04/Start”文件夹中找到，如果没有保存好上一章的项目文件，可以使用这些文件来开始本章的内容。

## 4.1 武器与炮塔

现在开始武器的详细设计。具体而言，关卡中包含了玩家和敌人的飞船。玩家必须具有射击敌人的功能，但是现在这些还无法做到，如图4.1所示。武器的详细设计需要注意3个重要的原则。第一，必须有一个发射器，当玩家按下发射按钮时就可以发射炮弹。第二，炮弹一旦产生就会独立地在场景中行动。第三，炮弹在与其他物体碰撞时具有伤害它们的能力。

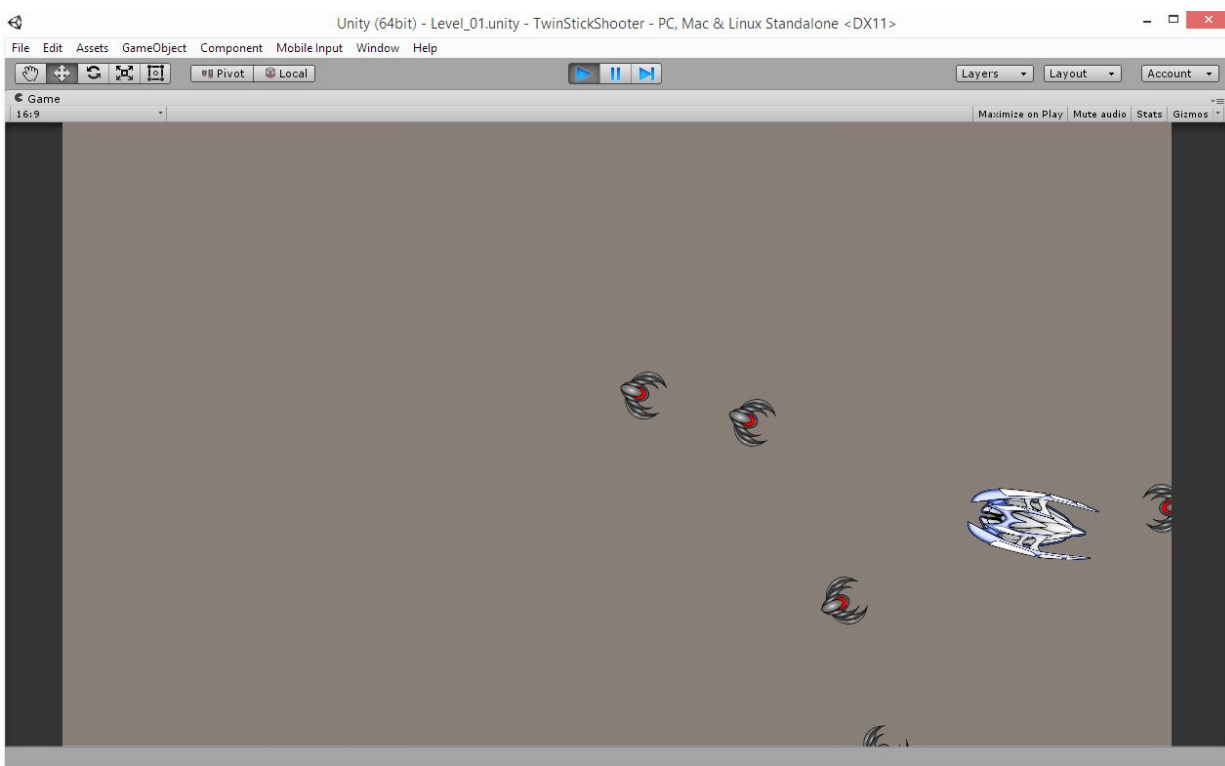


图4.1 目前为止的游戏

现在一步步地来实现这些功能，首先从炮塔产生和发射弹药的点开始。对于这个游戏，玩家只需要一个炮塔，不过一个更有吸引力的游戏中应该支持更多的炮塔，这样玩家就可以实现双重甚至多重的火力。下面创建第一个炮塔，首先在应用程序菜单上选中“GameObject | Create Empty”向场景中添加一个空的游戏对象，将其命名为“Turret”。然后将炮塔（Turret）对象放置到玩家飞船的前方，确定蓝色的前向向量箭头所指向的就是发射的炮弹的运行方向。最后，将炮塔（Turret）对象设置为玩家飞船的子对象，这可以通过将炮塔（Turret）对象拖曳到层次（Hierarchy）面板的玩家（Player）对象下方（见图4.2）。

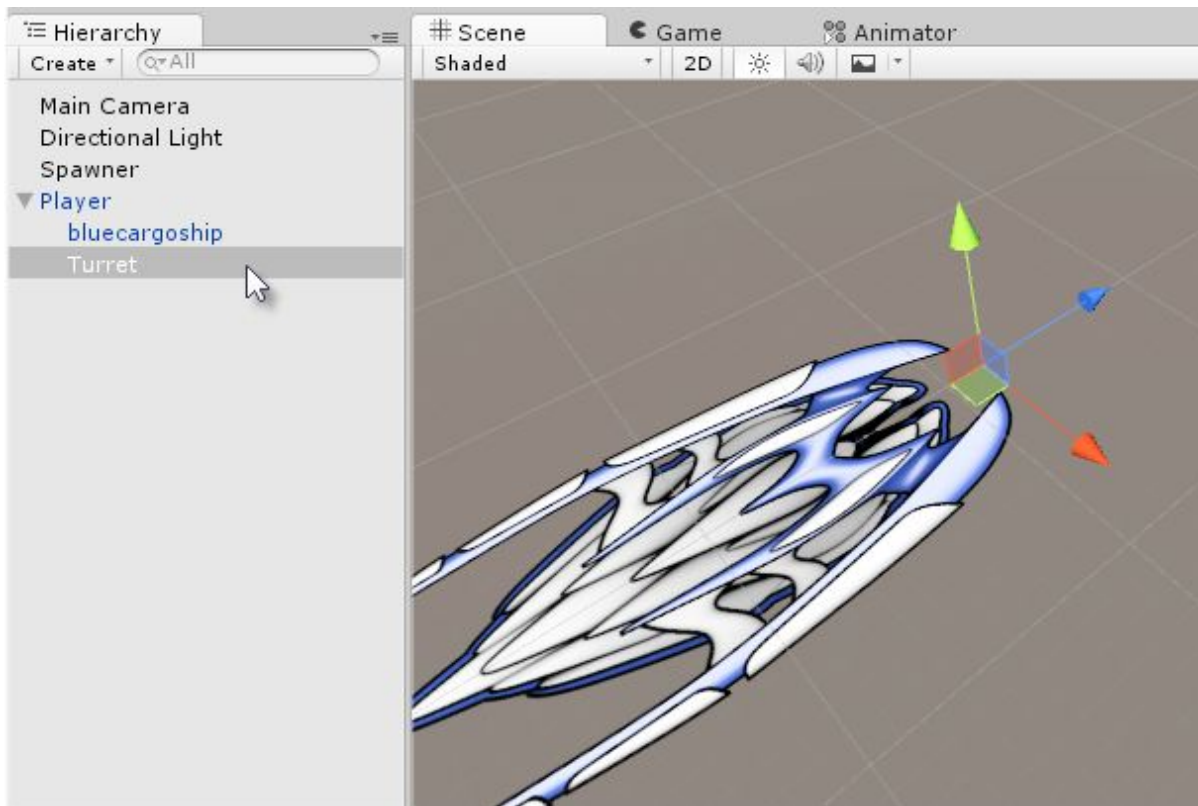


图4.2 将炮塔（Turret）对象作为玩家飞船的一个子对象

接下来创建一个发射炮弹的炮塔（**Turret**）对象，所有的炮弹都从这个位置发射出来。但是如果真的要产生一些炮弹，还需要设计一个炮弹对象。具体地说，需要创建一个炮弹（**Ammo**）预设体，在需要的时候就可以将这个预设体实例化为炮弹。下面就来实现这个功能。

## 4.2 炮弹预设体

当玩家在游戏进行中按下开火键时，宇宙飞船就应该在场景中发射炮弹对象。这些炮弹对象都是基于**Ammo**预设体的。现在就来创建这个预设体，首先将这个预设体的`texture`配置为炮弹的贴图。在项目（**Project**）面板上打开“**Textures**”文件夹，然后选中“**Ammo Texture**”。这个贴图包含了多个不同版本炮弹的图像精灵，这些炮弹图像精灵按照图4.3所示排成一行。当炮弹发射的时候，并不需要显示完整的贴图；相反，只需要其中的一个图像，或者将一些图像逐帧地组成动画序列。

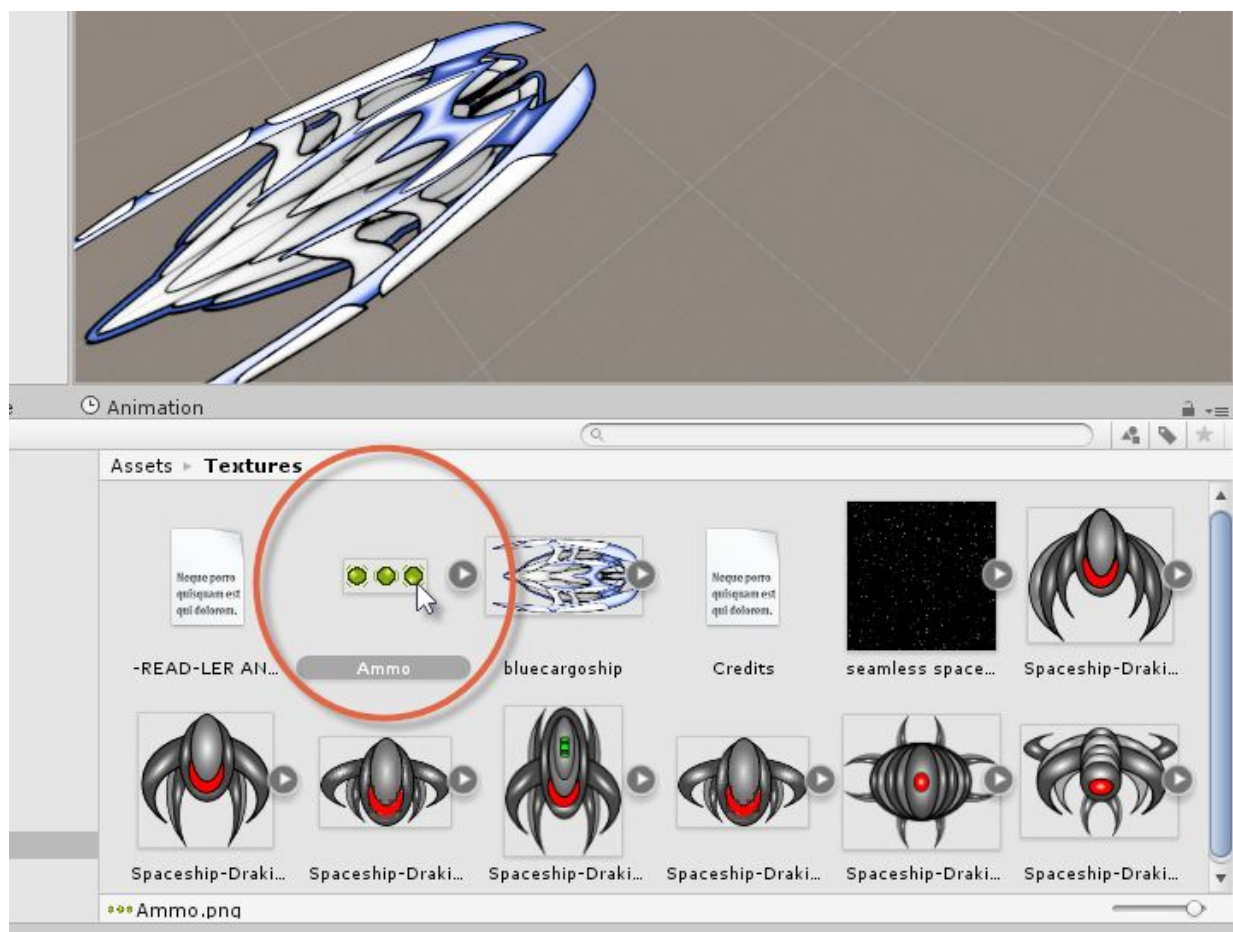


图4.3 创建炮弹预设体的准备工作

现在，Unity会将炮弹的每个部分贴图看作是一个整体，可以使用Sprite编辑器将这些部分分离开来。首先，在项目中选中贴图，之后（在对象Inspector面板）将“Sprite Mode”的值由“Single”修改为“Multiple”，表示在这个贴图中包含了多个图像精灵，如图4.4所示。

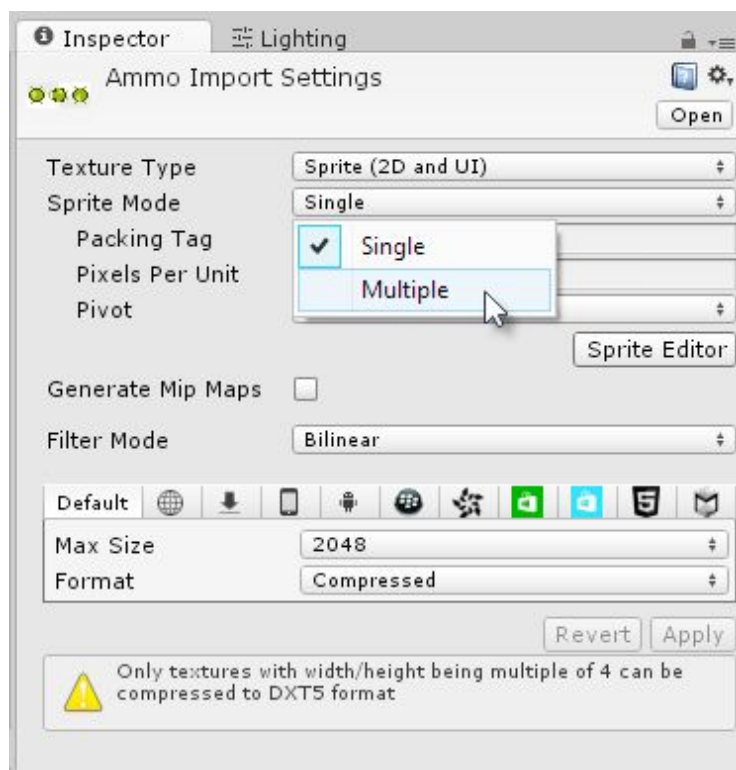


图4.4 贴图“Sprite Mode”设置为“Multiple”

完成上述操作之后单击“**Apply**”按钮，然后在对象Inspector面板中单击“**Sprite Editor**”按钮，就会打开**Sprite**编辑器，可以独立地编辑每一个图像精灵。首先单击并拖动鼠标来选择每个图像精灵，确保中心点与对象的中心对齐，如图4.5所示。之后，使用鼠标单击“**Apply**”按钮来保存这些改变。



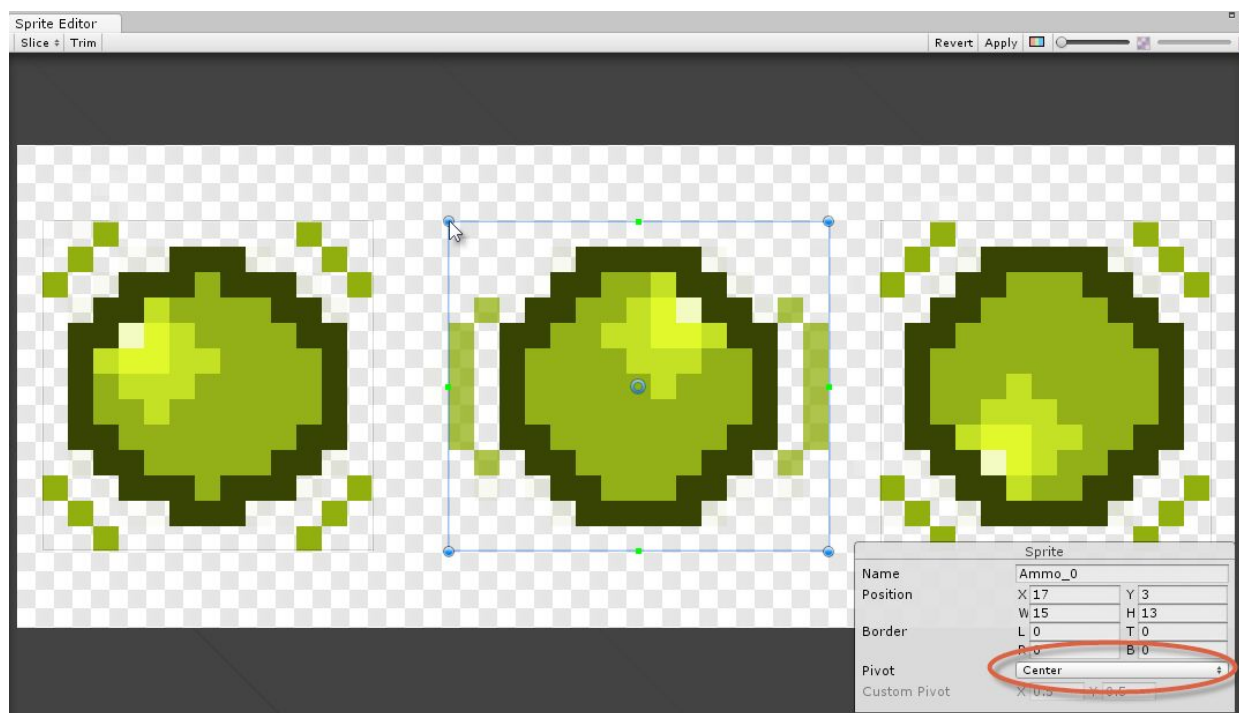


图4.5 在Sprite编辑器中将多个图像精灵分离

在Sprite编辑器中保存这些修改之后，Unity会自动地将相关图像精灵分割成独立的部分，这些部分都可以在项目（Project）面板中作为独立的对象选中。单击贴图向右的箭头，它所包含的所有图像精灵就会向外展开显示出来，如图4.6所示。

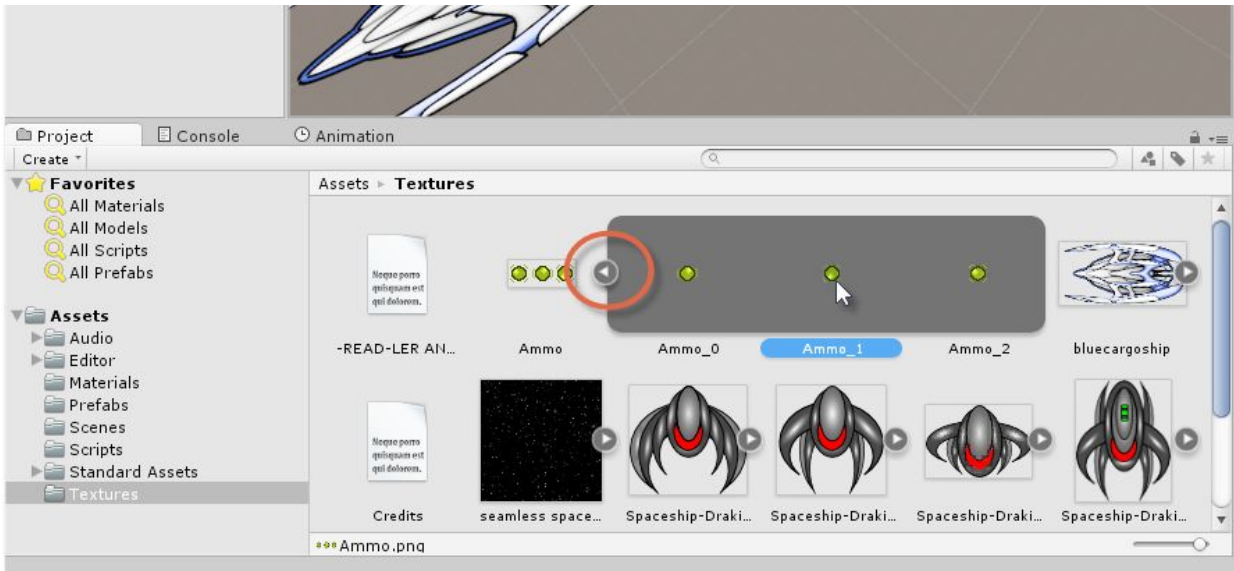


图4.6 展开贴图中的所有图像精灵

现在，将其中一个图像精灵从项目（**Project**）面板拖曳到场景中，它将会被添加成为一个图像精灵对象。这就是炮弹预设体的初始阶段。这个图像精灵在最开始的时候可能并不是直接面对着游戏摄像机，解决的方法就是将图像精灵旋转90度，直到它看起来的位置合适了为止，如图4.7所示。

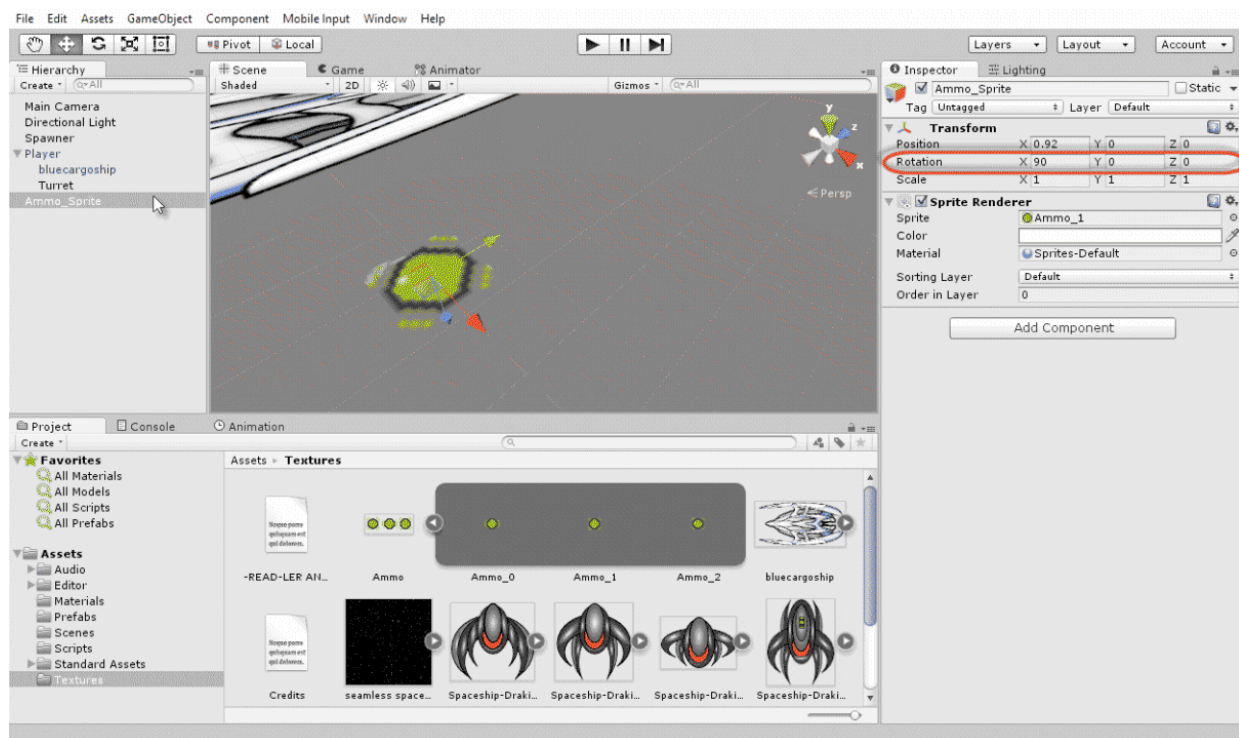


图4.7 将炮弹图像精灵进行对齐

此时在场景中创建了一个新的空游戏对象（从应用程序菜单中依次选中“GameObject | Create Empty”），并将其重命名为“Ammo”。将这个新的对象作为“Ammo\_Sprite”的父对象（见图4.8），以此来确保它的前向向量所指向的方向正是炮弹前进的方向，在炮弹上再次使用Mover脚本（在上一章中用到的），这个脚本可以控制炮弹的运动。

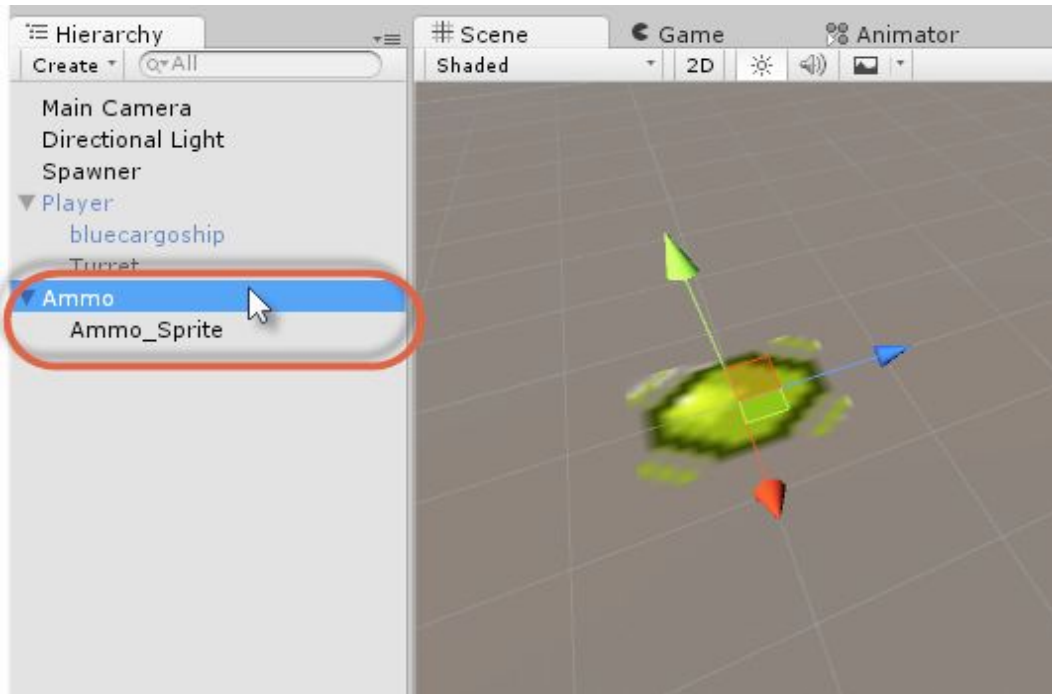


图4.8 创建一个炮弹对象

将Mover.cs从项目（Project）面板拖曳到“Ammo”父对象上，这样就可以将Mover作为“Ammo”的一个组件。然后在对象检查

（Inspector）面板中选中“Ammo”对象，将Mover组件中的“Max Speed”的值修改为7。最后向这个对象添加一个与其体积相仿的盒子碰撞器（从应用程序菜单依次选中“Component | Physics | Box Collider”）。下面可以对游戏进行测试了。“Ammo”对象向前开火就如同一个武器一样。如果炮弹运动的方向不正确，就需要确认父对象的蓝色前向向量指向的方向是否正确，如图4.9所示。

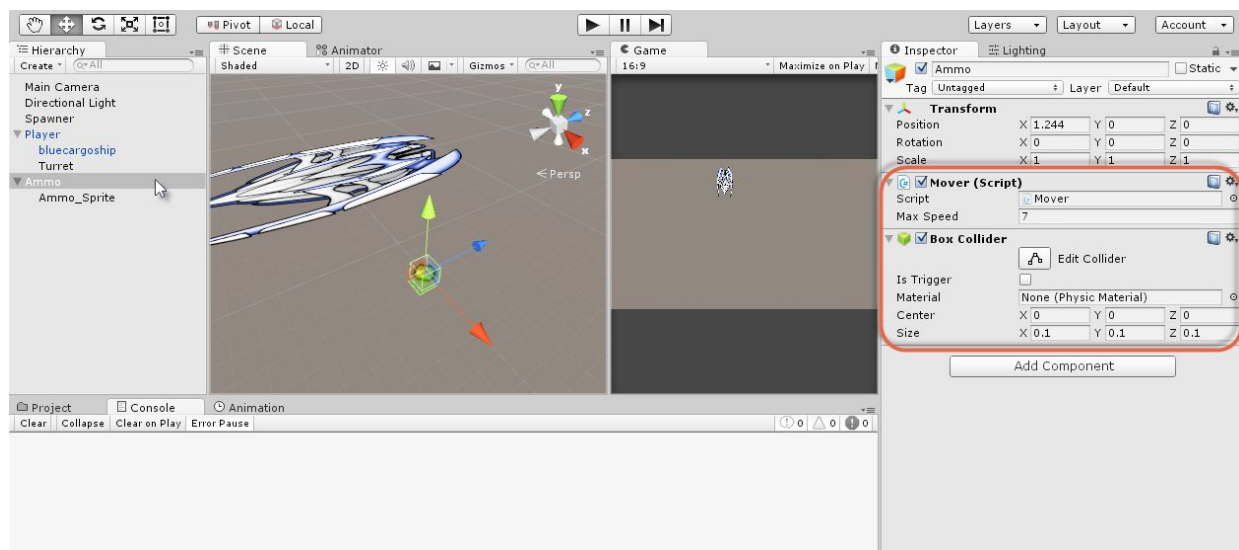


图4.9 向前运动的炮弹预设体（Mover组件和Collider组件）

接下来，向炮弹添加一个刚体组件，使它成为Unity物理体系中的一部分。首先选中炮弹（Ammo）对象，然后依次在应用程序菜单上选中“Component | Physics | Rigidbody”，在对象检查（Inspector）面板中的刚体组件处取消“Use Gravity”处的选中状态，这样在游戏进行中炮弹就不会掉落到地上。按照游戏设计思路，炮弹不应该受到重力的影响，而是沿着轨迹运动直到最终被销毁。这也表明了游戏开发时要把握的一个重要原则，不需要将现实世界中的物理情形毫无差池地应用到游戏中的每一个对象上，只需要能让游戏对象表现出合适行为的足够物理特性，如图4.10所示。

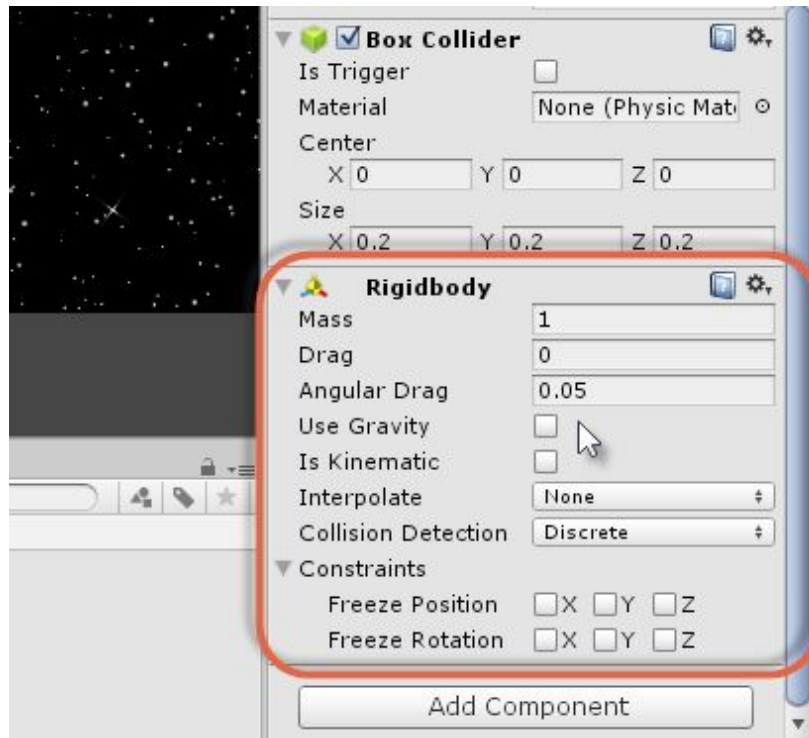


图4.10 将重力属性从炮弹对象上去除

除了向炮弹添加一个**Mover**脚本和物理属性之外，还需要给炮弹添加特有的行为。具体来说，当它与其他对象发生碰撞时，会对目标造成伤害，同时也会摧毁自身。为了实现这一功能，必须创建一个新的脚本文件，将这个新的文件命名为**Ammo.cs**。这个脚本中的全部代码如代码示例4.1所示。

#### 代码示例4.1:

```
//-----
using UnityEngine;
using System.Collections;
//-----
public class Ammo : MonoBehaviour
{
    public float Damage = 100f;
    public float LifeTime = 2f;
    //-----
    void OnEnable()
```



```

{
    CancelInvoke();
    Invoke("Die", LifeTime);
}
//-----
// Update会在每一帧调用一次
void OnTriggerEnter(Collider Col)
{
    //获取health组件
    Health H = Col.gameObject.GetComponent();

    if(H == null)return;

    H.HealthPoints -= Damage;
}
//-----
void Die()
{
    gameObject.SetActive(false);
}

//-----
}
//-----

```

下面对代码示例4.1进行总结。

- **Ammo**类应该附加到“**Ammo prefab**”对象上，所有的炮弹对象在创建时都应该将这个类实例化。这个类的目的就是当发生碰撞时对其他对象造成伤害。
- 当炮弹接触到一个移动单元（例如**Player**对象或者**Enemy**对象）上所附加的触发器时，就会调用**OnTriggerEnter**函数。然后它会查找附加在这个移动单元上的生命值组件，如果找到，就会将其生命值减少，减少的值为炮弹伤害值。生命值组件已经在上一章中创建过。
- 要注意每一个炮弹对象都有一个生命时间。这个时间代表了从炮塔开火和这个炮弹对象产生之后的生存时间。当生存时间到期后，炮弹就应该被销毁或者 停用。

- 函数Invoke用来在经过LifeTime时间间隔后停用炮弹对象，这个函数是在OnEnable函数中调用的。每当一个对象被激活之后，Unity就会自动调用OnEnable函数（也就是说，状态从Disabled转换到Enabled）。

将“Ammo”脚本文件从项目（Project）面板中的Scripts文件夹中拖曳到“Ammo”对象上，然后再将场景中整个“Ammo”对象拖曳回项目（Project）面板中的Prefabs文件夹以创建一个新的“Ammo prefab”，如图4.11所示。

现在已经创建了一个炮弹预设体，这个预设体可以从武器中产生，直接去攻击敌人。但是，现在并没有设计预设体的产生过程，接下来将对这部分内容进行处理。

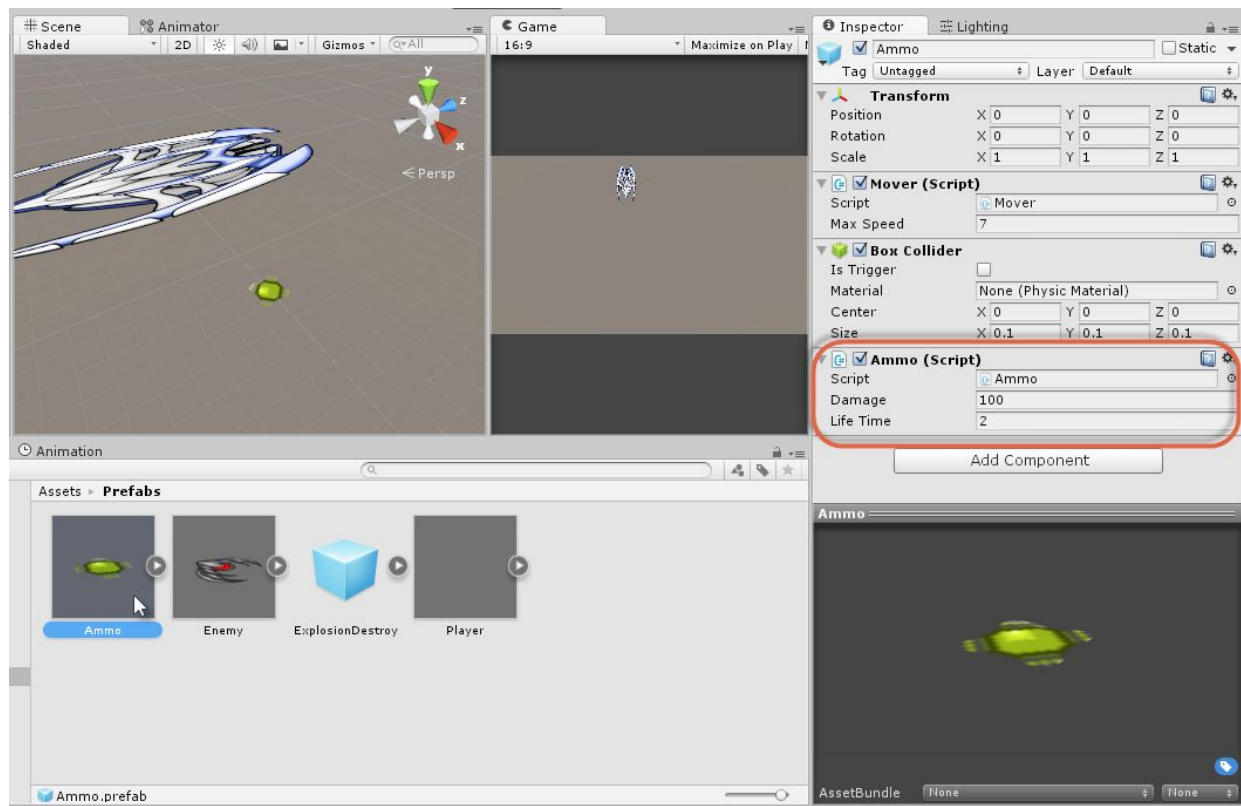


图4.11 创建一个炮弹预设体

## 4.3 炮弹的产生

到目前为止，炮弹预设体的创建留下了一个技术性的问题，如果没有谨慎地处理这个问题，将会对游戏造成严重的性能影响。具体来说，当宇宙飞船上的武器发射时，就会产生炮弹发射，击中并摧毁场景中的敌人。这的确不错，但是问题是玩家在进行游戏的时候可能会多次地按下开火按钮，甚至可能会长时间按住开火按钮不放，从而产生了数以百计的炮弹。可以使用实例化函数来动态地产生这些炮弹预设体，但是这些实例化计算时要花费大量的资源。当连续产生大量的物品时，通常会引起一个十分巨大的系统延迟，从而导致FPS降低到一个无法接受的程度，需要避免这种情况。

这种情况的解决方案就是Pooling技术，“Object Pooling”或者“Object Caching”。实质上，就是必须要产生一个大的且可以循环利用的炮弹对象池。最开始的时候这个炮弹对象池是隐藏的或者说是停用的，只有在需要的时候（例如当玩家发射武器时），才会激活对象。当炮弹与敌人相碰撞时或者炮弹的生命周期结束时，并不会彻底销毁该炮弹对象，而是将其关闭，然后放回到炮弹对象池中，如果再次需要使用，就可以直接使用对象池中的炮弹。利用这种工作方式可以避免对Instantiate的调用。在开始编程实现这个功能之前，先来完成一个AmmoManager类。这个类将实现两个功能：第一，在场景启动的时候生成一对象池的炮弹，第二，当需要使用炮弹时，例如使用武器开火，就会从对象池中提供炮弹对象。下面的代码示例4.2中AmmoManager类实现这个功能。

## 代码示例4.2:

```
//-----
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
//-----
public class AmmoManager : MonoBehaviour
{
    //-----
    //对 ammo预设体的引用
    public GameObject AmmoPrefab = null;

    //Ammo池计数
    public int PoolSize = 100;

    public Queue<Transform> AmmoQueue = new Queue<Transform>();

    //产生一个ammo对象的队列
    private GameObject[] AmmoArray;

    public static AmmoManager AmmoManagerSingleton = null;
    //-----
    // 初始化函数
    void Awake ()
    {
        if(AmmoManagerSingleton != null)
        {
            Destroy(GetComponent<AmmoManager>());
            return;
        }

        AmmoManagerSingleton = this;
        AmmoArray = new GameObject[PoolSize];

        for(int i=0; i<PoolSize; i++)
        {
            AmmoArray[i] = Instantiate(AmmoPrefab, Vector3.zero,
                Quaternion.identity) as GameObject;
            Transform ObjTransform =
                AmmoArray[i].GetComponent<Transform>();
            ObjTransform.parent = GetComponent<Transform>();
            AmmoQueue.Enqueue(ObjTransform);
            AmmoArray[i].SetActive(false);
        }
    }
    //-----
    public static Transform SpawnAmmo
        (Vector3 Position, Quaternion Rotation)
```

```

{
    //获取 ammo
    Transform SpawnedAmmo =
        AmmoManagerSingleton.AmmoQueue.Dequeue();

    SpawnedAmmo.gameObject.SetActive(true);
    SpawnedAmmo.position = Position;
    SpawnedAmmo.localRotation = Rotation;

    //添加到Queue尾部
    AmmoManagerSingleton.AmmoQueue.Enqueue(SpawnedAmmo);

    //返回ammo
    return SpawnedAmmo;
}
//-----
}
//-----

```

下面对代码示例4.2进行总结。

- **AmmoManager**类包含了一个**AmmoArray**成员变量，这个变量中包含了所有炮弹对象的列表（顺序引用数组），这些炮弹对象都会在游戏启动的时候产生（在**Awake**函数中实现）。
- **AmmoArray**的大小要与**PoolSize**相匹配。这个大小指的就是要产生的炮弹对象的数量。函数**Awake**会在关卡开始的时候产生炮弹对象，这些对象都会被**Enqueue**添加到队列中。
- 当这些炮弹对象产生之后，每一个炮弹对象都会被**SetActive(false)**设置为停用状态，并存储在对象池中，直到需要使用的时候。
- **AmmoManager**使用**Mono**库中**Queue**类来确定当开火键按下时，如何从对象池中选中炮弹对象。**Queue**的工作机制是先进先出（**First In First Out, FIFO**）。换句话说，每次将一个炮弹对象添加到**Queue**中，当炮弹对象被选中激活之后就可以从**Queue**中移除。移除的对象总是位于**Queue**的最前端。关于**Queue**类的更多信息可以在下面

的网址找到：<https://msdn.microsoft.com/en-us/library/7977ey2c%28v=vs.110%29.aspx>。

- 在Awake函数中，Queue对象会调用Enqueue函数向Queue中逐个添加对象。
- 调用SpawnAmmo函数产生场景中新的炮弹。这个函数中不需要使用Instantiate函数，而是使用Queue对象来代替。它将Queue中的第一个炮弹对象移出，并激活它。然后再次将它添加到Queue中所有的炮弹对象的后面。通过这种方法，就可以循环地使用所有的炮弹对象。
- AmmoManager被编写为一个单态对象，这就意味着，在任何时候都只有一个对象的实例存在于场景中。这个功能是通过静态成员AmmoManagerSingleton实现的。关于单态对象的更多信息，可以访问<https://www.packtpub.com/game-development/mastering-Unity-5x-scripting>里Packet出版社出版的*Mastering Unity Scripting*一书。

在应用程序菜单处依次单击“GameObject | Create Empty”来在场景中创建一个新的游戏对象，并将其命名为AmmoManager。然后从项目（Project）面板处将AmmoManager脚本拖曳到场景中选中的对象上。创建成功之后，从Prefabs文件夹中将“Ammo prefab”拖曳到对象Inspector面板中“Ammo Manager”组件处的“Ammo Prefab”后面的位置处，如图4.12所示。



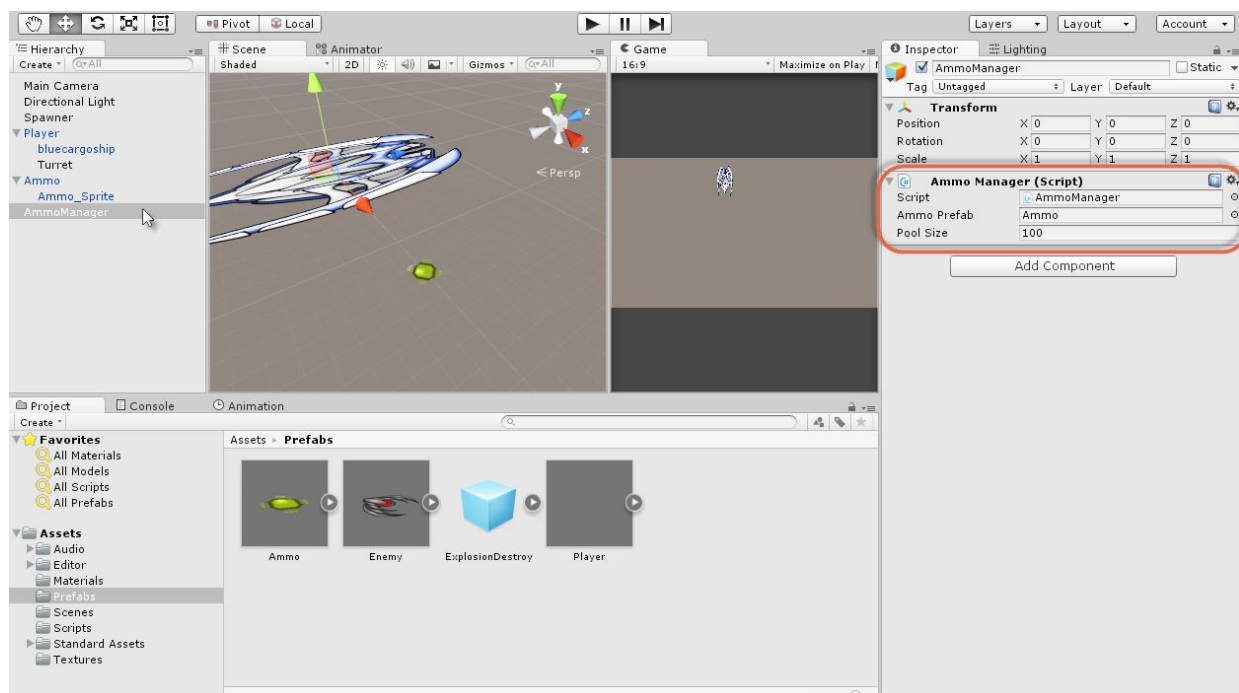


图4.12 向“Ammo Manager”添加一个对象

现在场景中已经包含了一个AmmoManager对象，这个对象实现了Ammo对象池。不过，当玩家按下开火键时仍然什么都没有发生，这是因为在现在的游戏中仍然缺乏玩家按键与产生炮弹之间的联系。这是因为没有编写这方面的代码。可以利用上一章中编写的PlayerController脚本来完成这两者之间的联系。现在来改进这段脚本，修改以后的脚本将具有产生炮弹的能力，重新编写过的PlayerController类内容如代码示例4.3所示，加粗的部分就是修改过的部分。

### 代码示例4.3:

```
//-----
using UnityEngine;
using System.Collections;
//-----
public class PlayerController : MonoBehaviour
{
    //-----
```

```

private Rigidbody ThisBody = null;
private Transform ThisTransform = null;

public bool MouseLook = true;
public string HorzAxis = "Horizontal";
public string VertAxis = "Vertical";
public string FireAxis = "Fire1";

public float MaxSpeed = 5f;
public float ReloadDelay = 0.3f;
public bool CanFire = true;

public Transform[] TurretTransforms;
//-----
//初始化函数
void Awake ()
{
    ThisBody = GetComponent();
    ThisTransform = GetComponent();
}
//-----
// Update会在每一帧调用一次
void FixedUpdate ()
{
    //对运动进行更新
    float Horz = Input.GetAxis(HorzAxis);
    float Vert = Input.GetAxis(VertAxis);
    Vector3 MoveDirection = new Vector3(Horz, 0.0f, Vert);
    ThisBody.AddForce(MoveDirection.normalized * MaxSpeed);

    //对速度进行限制
    ThisBody.velocity = new Vector3
        (Mathf.Clamp(ThisBody.velocity.x, -MaxSpeed, MaxSpeed),
        Mathf.Clamp(ThisBody.velocity.y, -MaxSpeed, MaxSpeed),
        Mathf.Clamp(ThisBody.velocity.z, -MaxSpeed, MaxSpeed));

    //鼠标控制
    if(MouseLook)
    {
        //对rotation进行更新 -转向去面对鼠标指针
        Vector3 MousePosWorld = Camera.main.ScreenToWorldPoint(new
            Vector3(Input.mousePosition.x,
Input.mousePosition.y,0.0f));
        MousePosWorld = new Vector3(MousePosWorld.x, 0.0f,
            MousePosWorld.z);

        //获得光标的方向
        Vector3 LookDirection = MousePosWorld -
            ThisTransform.position;
    }
}

```

```

        //对 rotation进行固定更新
        ThisTransform.localRotation = Quaternion.LookRotation
            (LookDirection.normalized,Vector3.up);
    }
    //检查开火控制
    if(Input.GetButtonDown(FireAxis) && CanFire)
    {
        foreach(Transform T in TurretTransforms)
            AmmoManager.SpawnAmmo(T.position, T.rotation);

        CanFire = false;
        Invoke ("EnableFire", ReloadDelay);
    }
}
//-----
void EnableFire()
{
    CanFire = true;
}
//-----
public void Die()
{
    Destroy(gameObject);
}
}
//-----

```

下面对代码示例4.3进行总结。

- PlayerController类中包含了一个名为TurretTransform的数组变量，这个数组变量中列出了所有可以作为炮塔产生地点的对象。
- 在Update函数中，PlayerController会检测开火键是否被按下。如果检测到这个键被按下，代码就会在所有的炮塔循环，并在每一个炮塔的位置产生一个炮弹。
- 当开火时，就会将ReloadDelay的值设置为“True”。这就表示只有当延迟时间（delay）完成以后，才可以再次开火。

当将这些代码添加到PlayerController类之后，在场景中选中Player对象，将空的炮塔对象拖曳到“Turret Transform”下面的槽位里。在这个

例子中只使用了一个炮塔，不过如果需要，也可以添加更多的炮塔，如图4.13所示。

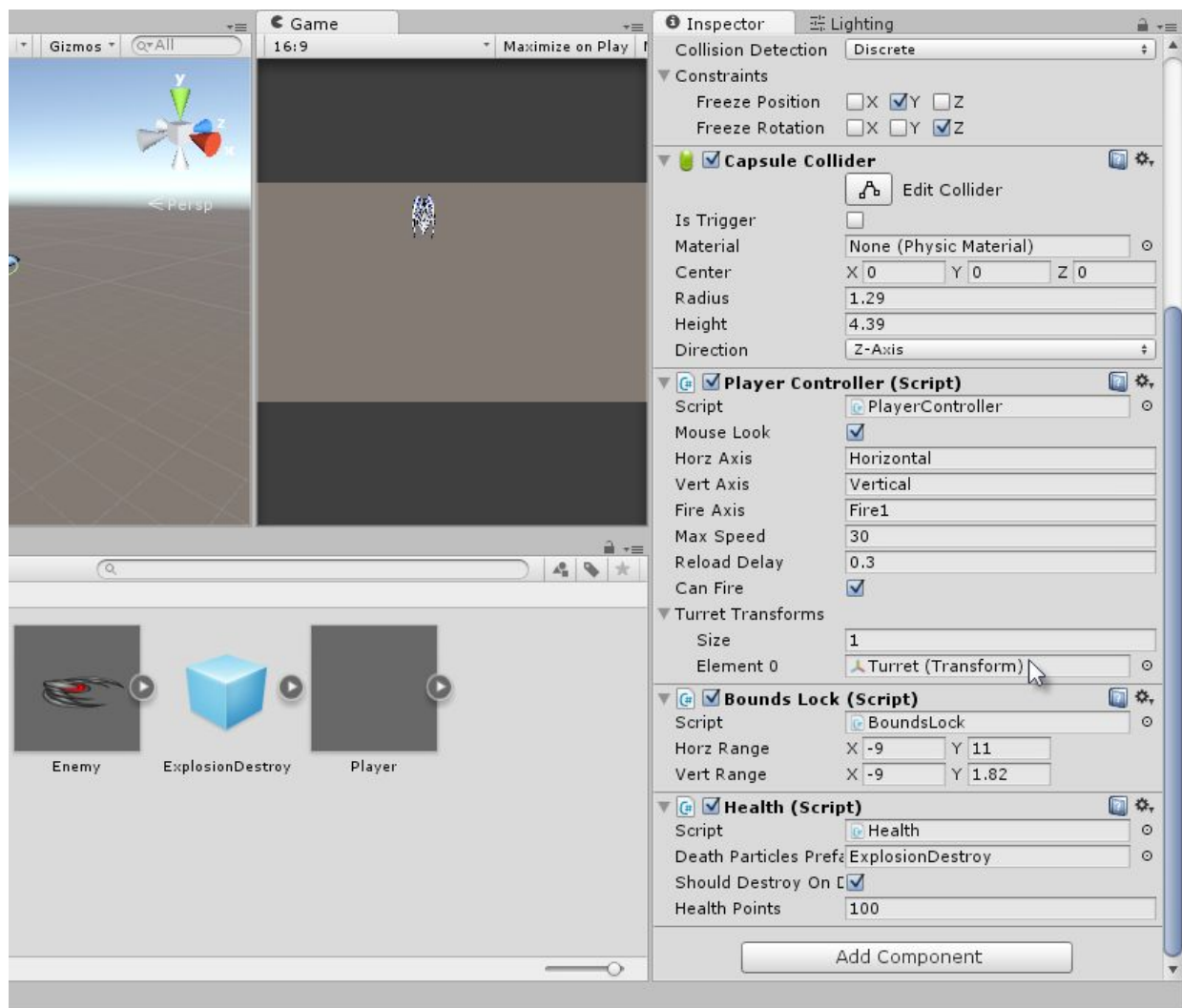


图4.13 为炮弹的产生配置“TurretTransform”属性

现在已经完成飞船开火的功能。在场景中进行游戏的测试，当按下键盘上的开火键或者鼠标左键时，炮弹就会产生。不过在这次的测试中，有两个主要的问题：第一，产生的炮弹可能太大或者太小；第二，产生的炮弹有时可能会与玩家宇宙飞船发生碰撞。接下来逐步对这些问题进行解决。

如果炮弹的大小不正确，只需要对预设体的尺寸进行修改即可。首先在项目（Project）面板上选中“**Ammo Prefab**”，从检查（Inspector）面板处的“**Transform**”组件处输入一个新的值，如图4.14所示。

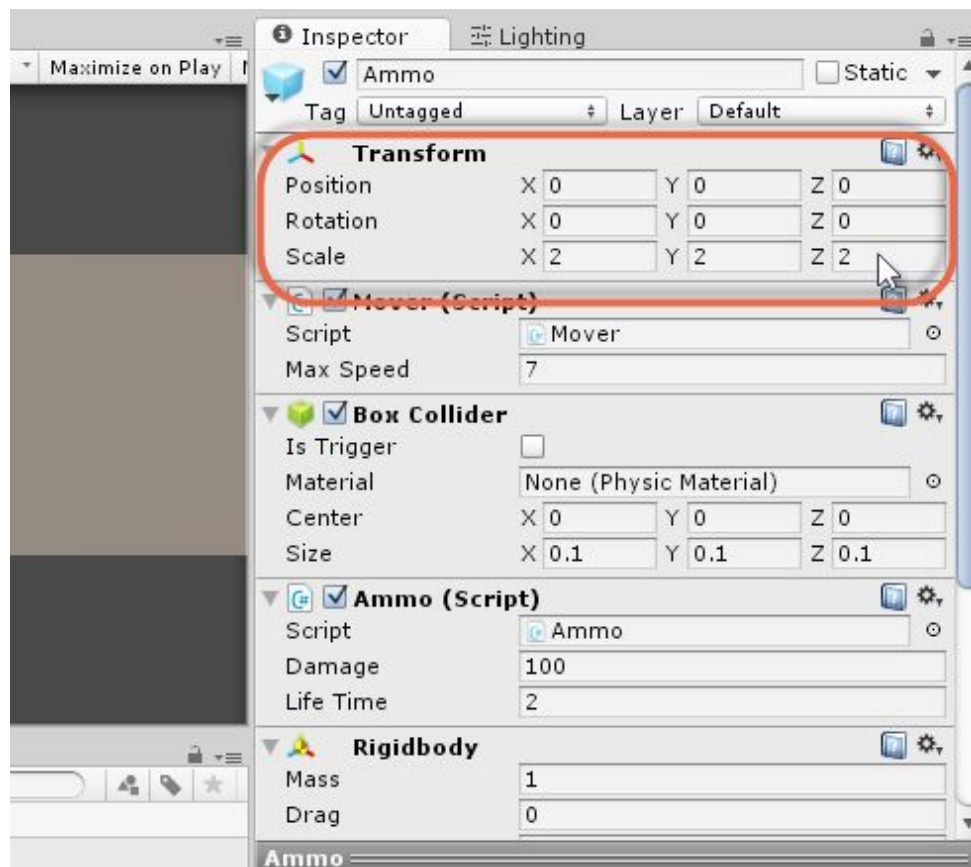


图4.14 修改“Ammo Prefab”的大小

如果炮弹出现了和玩家宇宙飞船相互碰撞的情形，就需要对炮弹进行设置，使得炮弹与玩家之间没有任何反应。为了实现这一点，可以使用物理层（**Layer**）。简而言之，玩家宇宙飞船和炮弹应该添加到同一个层，并且这一个层中的所有对象互相之间都是没有任何物理影响的。首先，在场景中选中**Player**对象，然后从对象检查（Inspector）

面板，单击“Layer”下拉列表框，然后从弹出的下拉菜单中选中“Add Layer”，如图4.15所示。

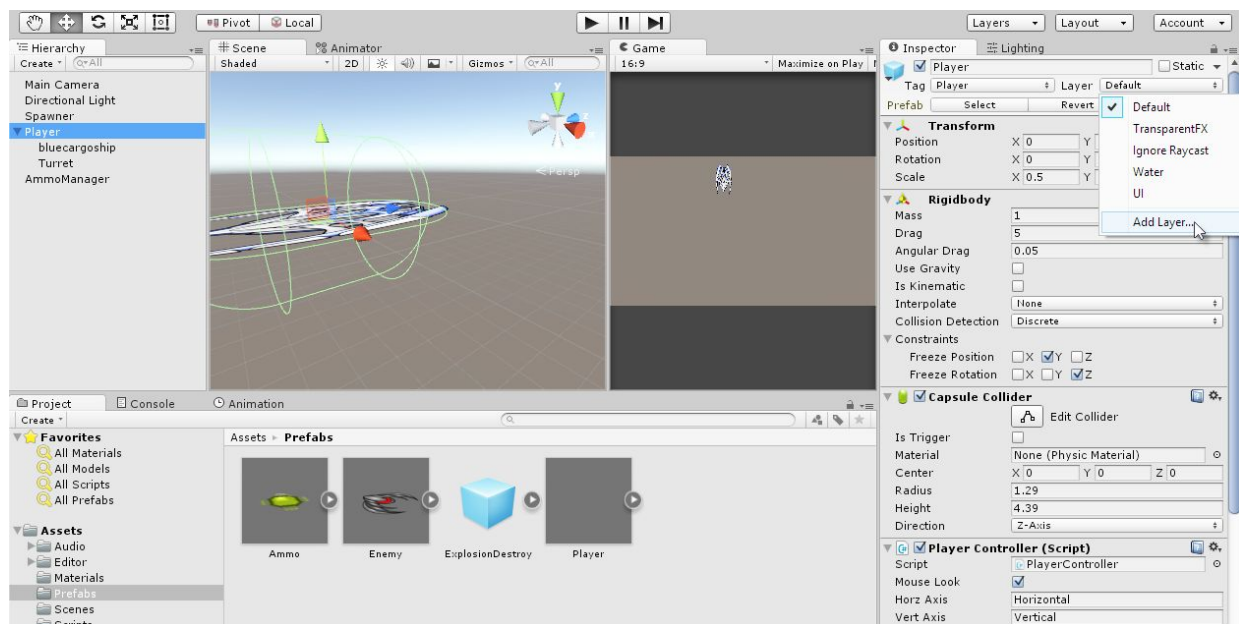


图4.15 创建一个物理无关的新层

将这个层命名为“Player”，表示在这个层中的所有对象都与“Player”对象有关，如图4.16所示。



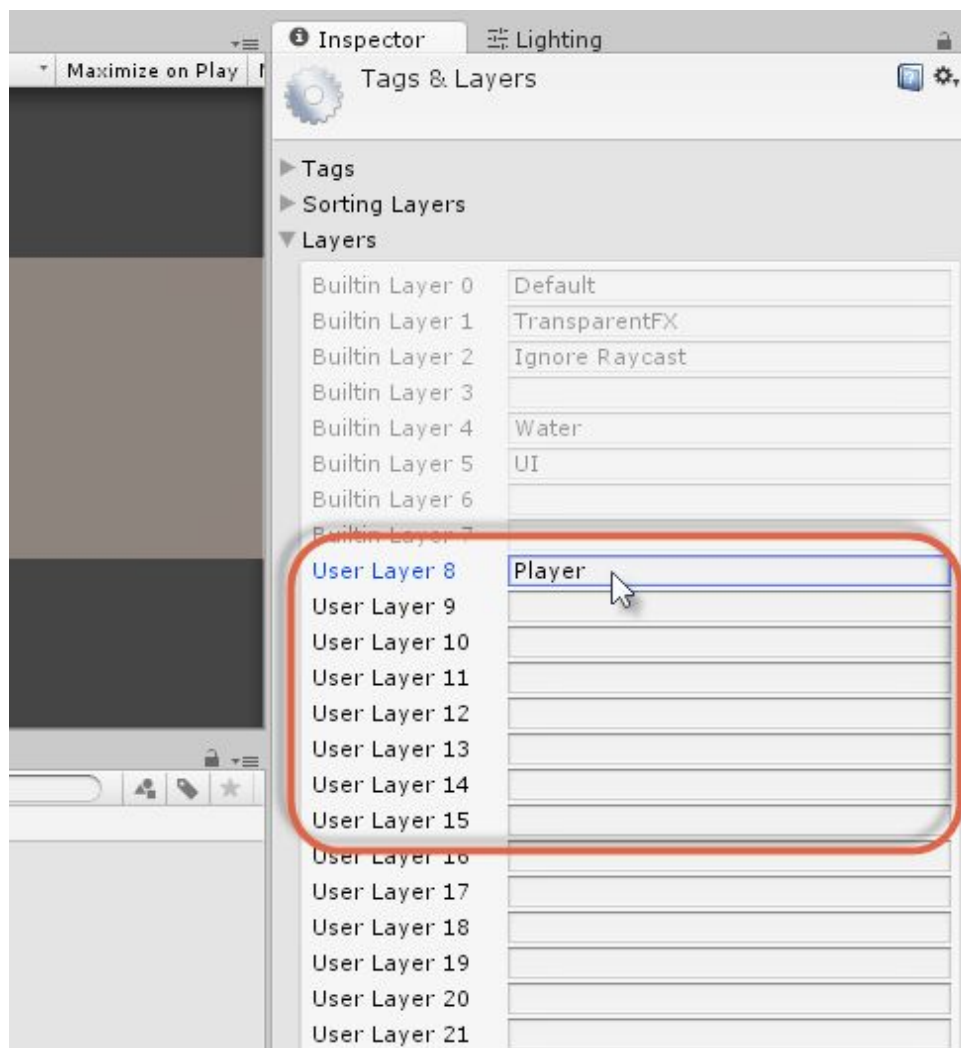


图4.16 创建一个新的层

将场景中的**Player**对象和项目（**Project**）面板中的“**Ammo Prefab**”都放置到新创建的**Player**层中。对这两个对象依次执行如下操作，首先选中“**Layer**”下拉列表框，然后选中其中的“**Player**”选项，如图4.17所示。如果此时弹出了一个对话框，同样要选择将子对象一同修改。这就可以确保所有的子对象与其父对象关联到了相同的层。

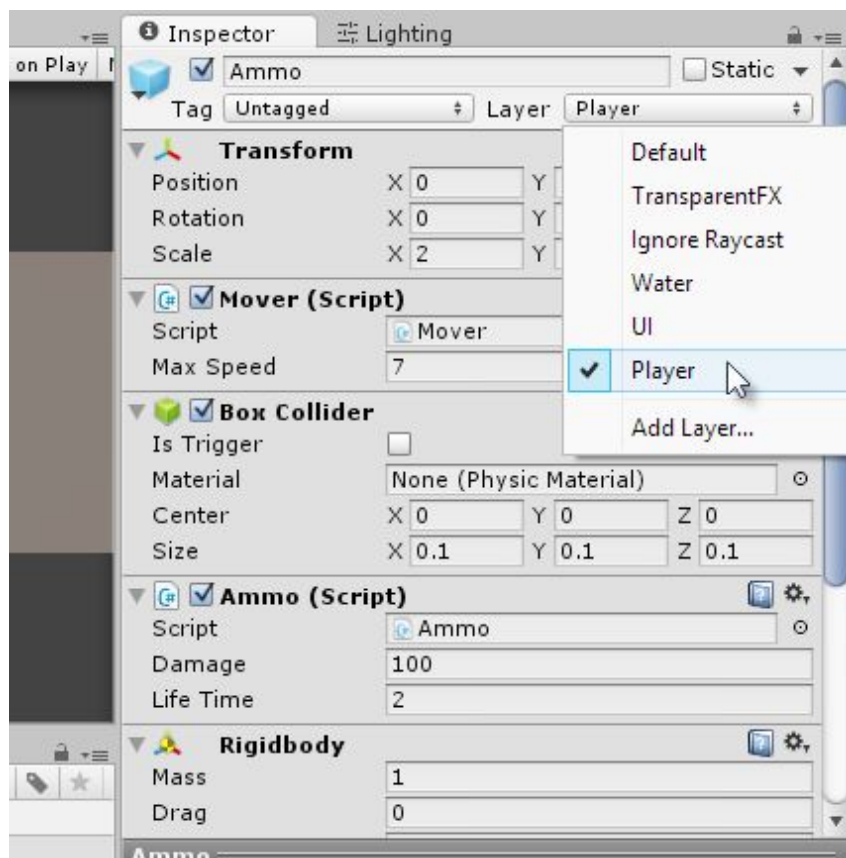


图4.17 将玩家（Player）和炮弹（Ammo）都放置到Player层

玩家（Player）对象和炮弹（Ammo）对象都已经添加到了相同的层，现在同一层中的所有对象都是可以设置相互无影响的。接下来，设置这些对象在物理属性上没有影响。为了实现这一点，在应用程序菜单上依次选中“Edit | Project Settings | Physics”，如图4.18所示。

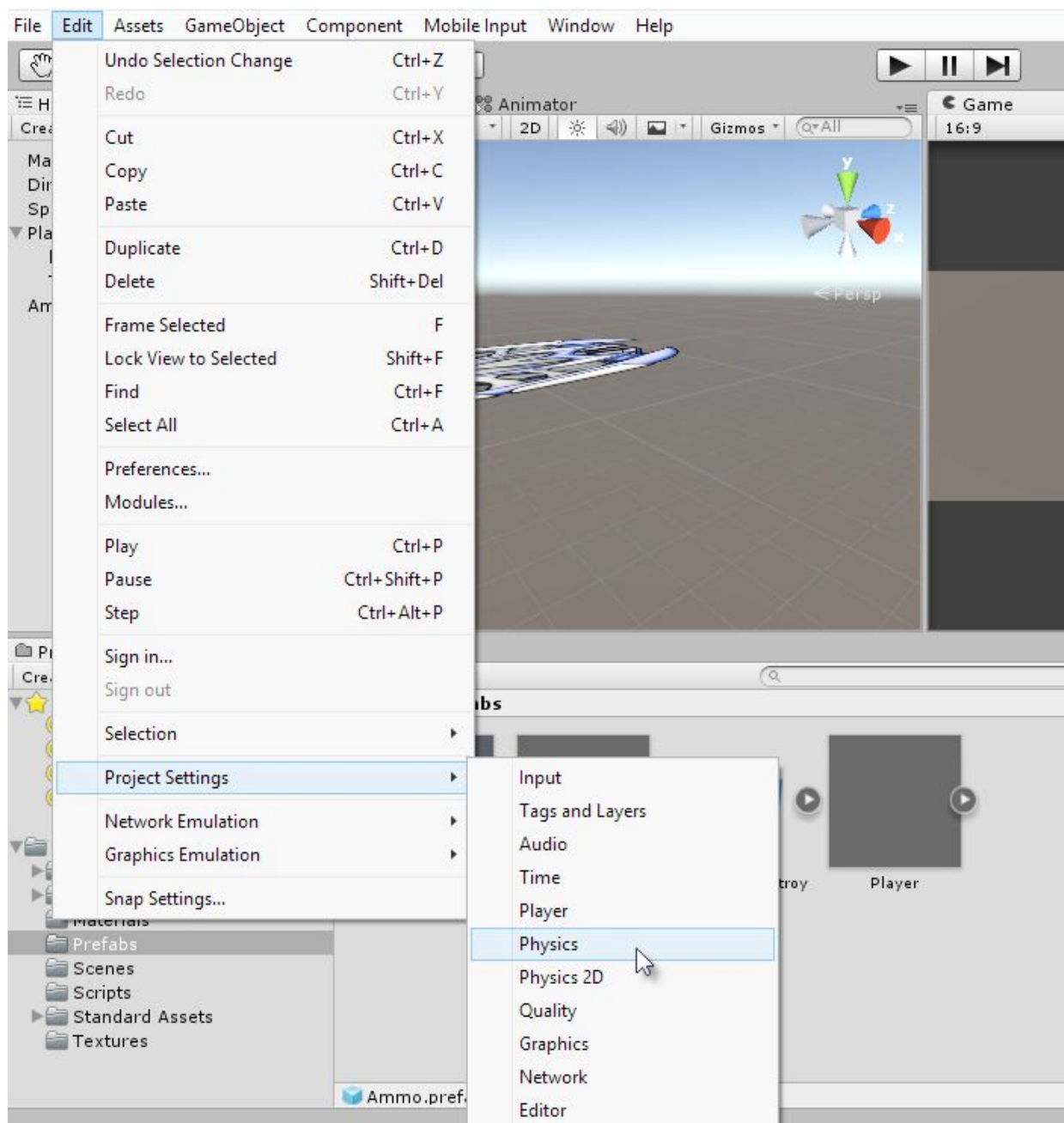


图4.18 选中“Physics”选项

在对象检查（Inspector）面板处有全局Physics的设置。在检查（Inspector）面板底部的“Layer Collision Matrix”组件处展示了各个层之间的相互影响。任意两个层交叉的复选框如果被选中，就表示这两个

层中的对象会相互影响。基于这个原理，将**Player**层后面复选框中的选中取消，阻止这一层里各个对象的碰撞，如图4.19所示。

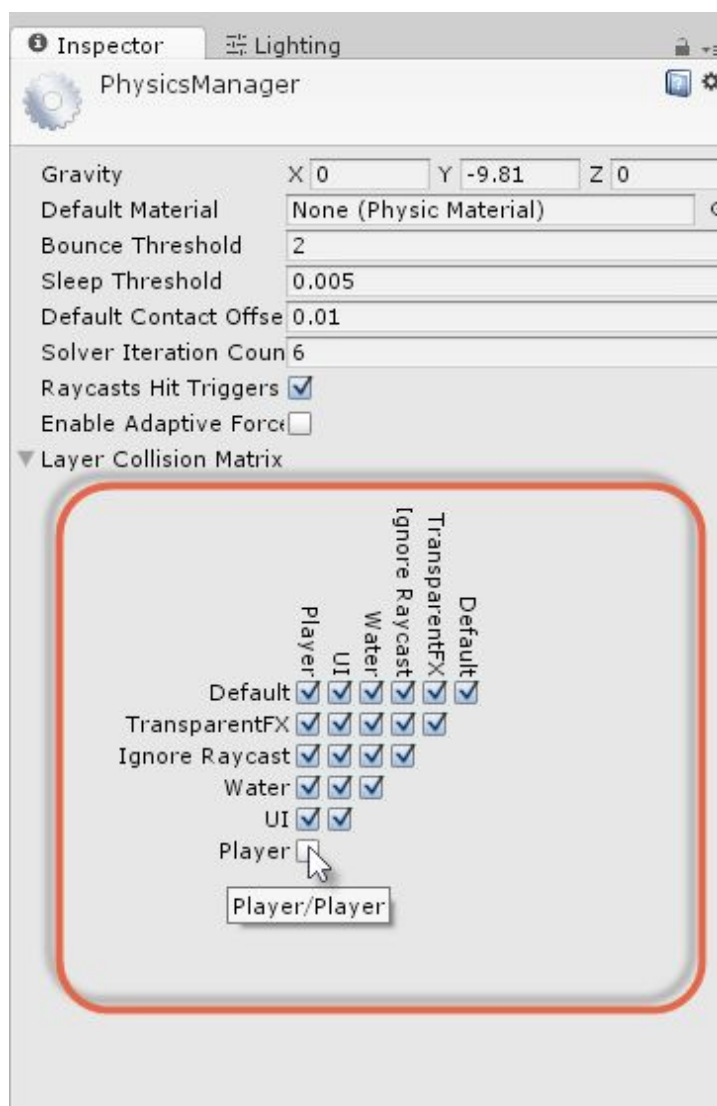


图4.19 对“Layer Collision Matrix”进行调整以改善碰撞

在对象检查（Inspector）面板中对“Layer Collision Matrix”进行设置，在工具栏上按下“Play”按钮对游戏进行测试。当开始对游戏进行测试后，按下开火键，炮弹就会在炮塔处产生，同时也不会再与宇宙飞船发生碰撞。产生的炮弹只会去碰撞和摧毁敌人，如图4.20所示。

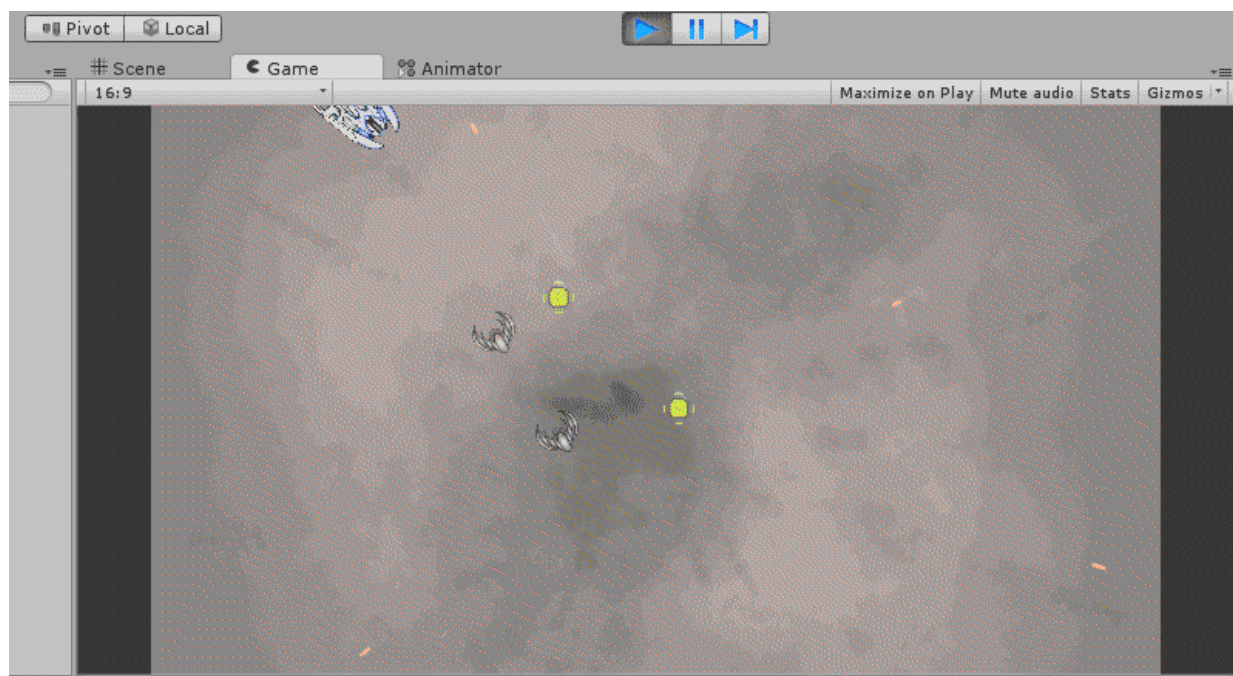


图4.20 通过开火消灭敌人

现在已经有了一艘可以开火并消灭敌人的宇宙飞船，而且它的物理属性和我们设计的一样。你可能想要自定义玩家的控制，或者想要使用一个游戏手柄来控制游戏。接下来的内容将完成这部分设计。

## 4.4 用户控制

可能你并不喜欢默认的游戏控制方式——上下、左右、开火，想要做出一些改变，这些输入会被函数`Input.GetAxis`（在之前出现过）所读取。这个函数的名字易于记忆，同时也隐藏了Unity中设备输入与虚拟轴之间的映射关系。接下来简单地看一下如何实现自定义。可以通过从应用程序菜单处依次选中“**Edit | Project Settings | Input**”来修改游戏的输入设置，如图4.21所示。



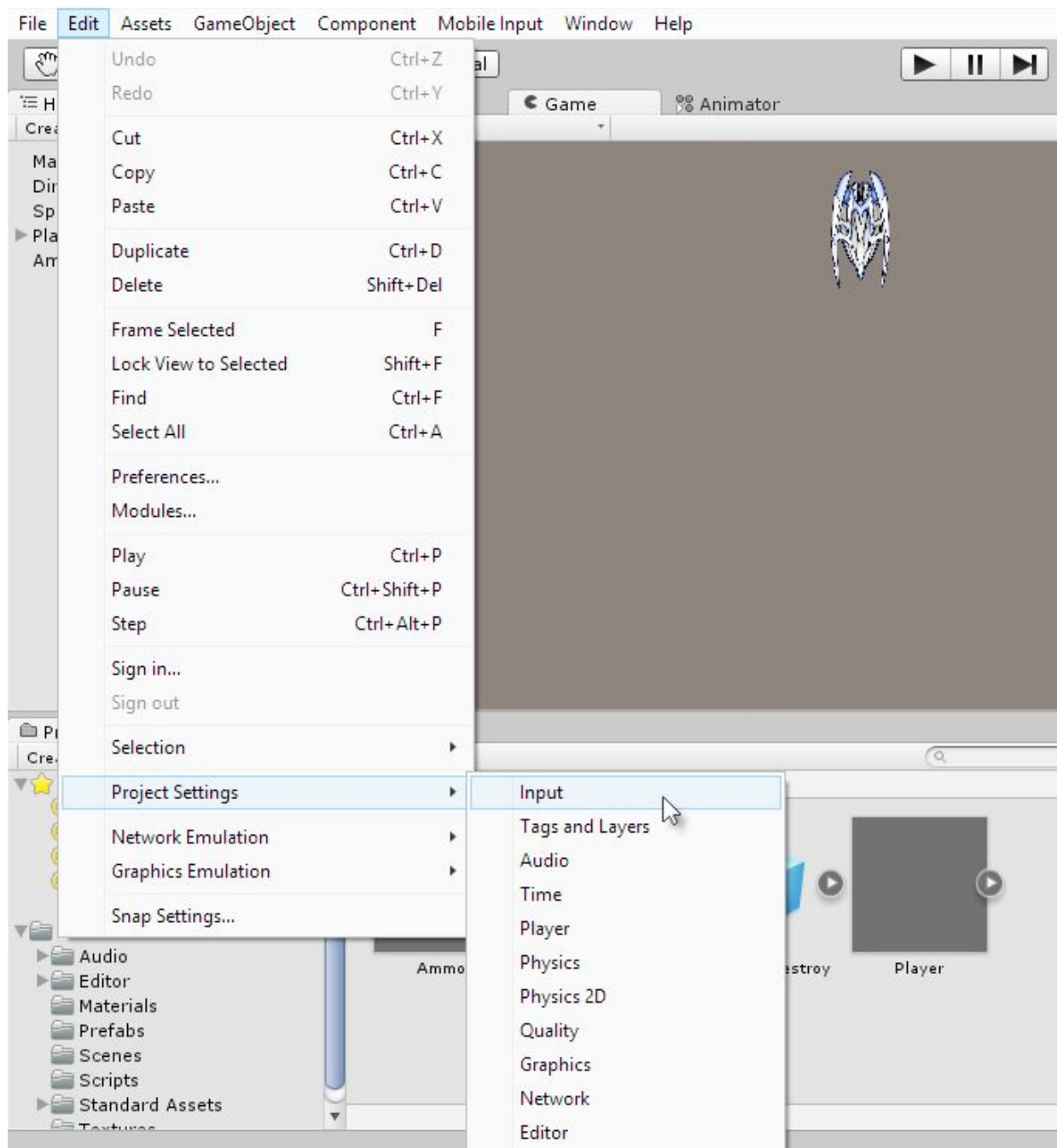


图4.21 访问“Input”选项

当选中了这个选项之后，在对象检查（Inspector）面板就会出现一个自定义输入轴集合，这个集合是以列表形式出现的，如图4.22所示。



这个列表可以定义输入系统的所有轴，“Horizontal”和“Vertical”两个轴都在这里显示出来。

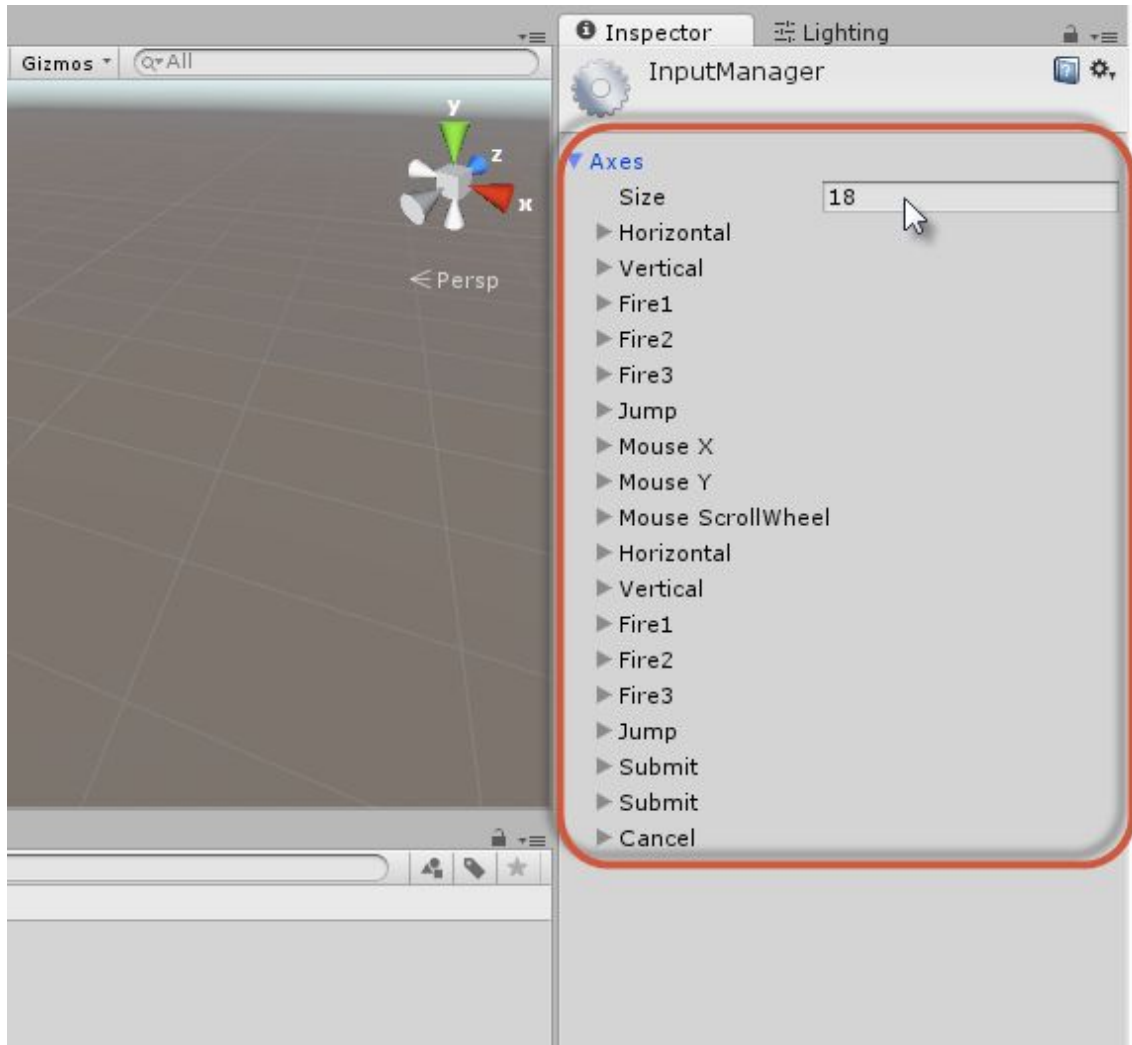


图4.22 输入轴的设置

通过在对象检查（Inspector）面板中对每个轴进行扩展，就可以简单地自定义用户输入的映射方式，即如何指定硬件设备上的键和控制，例如键盘和鼠标，都会映射成一个轴。例如，Horizontal轴就被定义了两次。对于第一个定义，Horizontal被映射成了左和右，以及键盘上的A和D，右和D被映射为正按钮，这是因为当按键被按下之后，它

们就会依靠Input.GetAxis函数产生正的浮点值，这个值为0~1中任意一个。左和A键被映射成为负的浮点数，这样就可以根据值的正负来确定对象移动的方向，如图4.23所示。

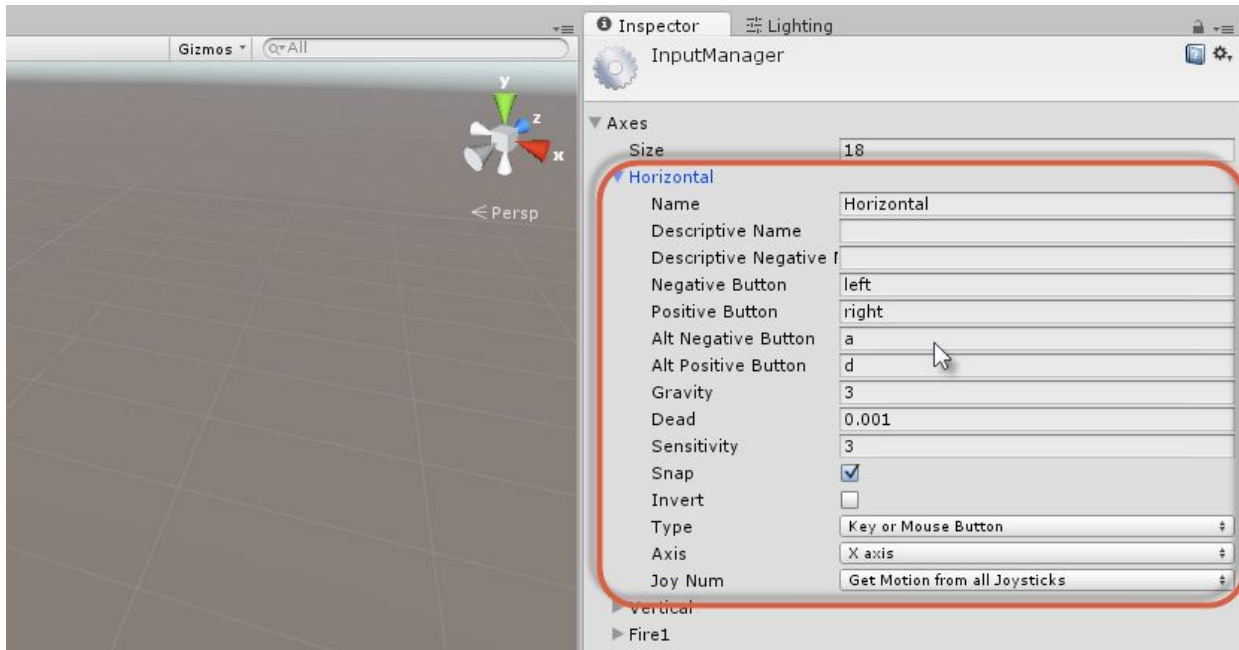


图4.23 对输入轴进行配置

同样，Horizontal也在对象检查（Inspector）面板中被定义了两次，一次在整个列表的上方位置，一次在整个列表的下方位置。这两次定义的效果是叠加的，它们之间并不矛盾。这种方式允许将多个设备映射到同一个轴上面，这样游戏就可以实现在多平台和多设备上面运行。默认情况下，Horizontal映射的第一个定义是键盘上的左方向键、右方向键、“A”和“D”键，第二个定义是操纵杆的运动。这两个定义都是有效的，它们协同工作。如果需要，可以对同一个轴进行更多的定义，这主要取决于所要支持的设备，如图4.24所示。

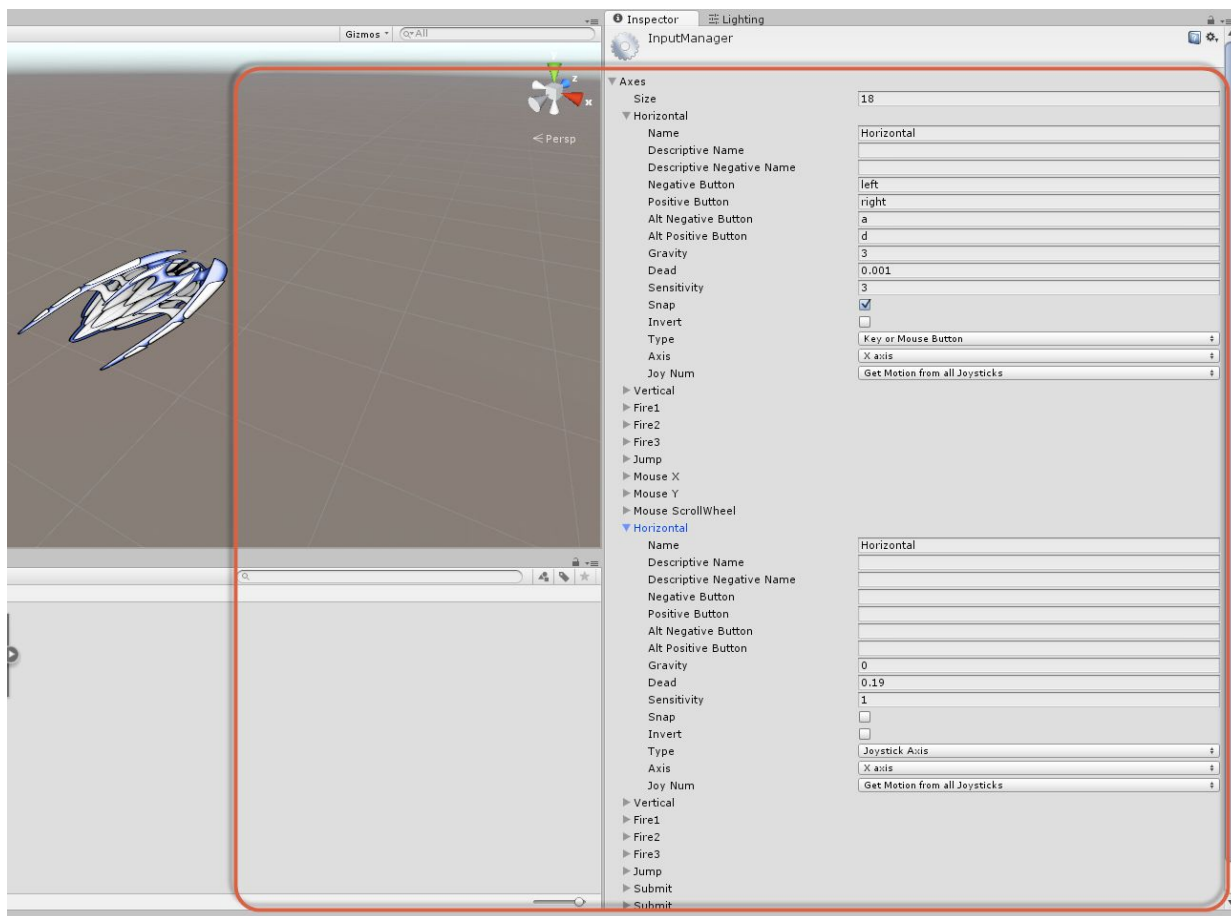


图4.24 定义两次“Horizontal Axes”

对于这个项目，保留了默认的控制模式，不过如果希望支持多种不同的配置方式，也可以修改或者添加新的控制模式。关于用户输入和自定义控制的更多信息可以在Unity的在线文档

<http://docs.Unity3d.com/Manual/classInputManager.html>中查看。

## 4.5 分数和评分——UI和文本对象

下面继续游戏的设计，现在开始计分系统的实现，为此创建了一个名为GameController的类。这个类用来管理游戏全局范围和总体性的行为，同样包括了比分。因为对于这个游戏来说，分数指的是玩家所

获得的一个独立的全局性进度的标志。在开始计分系统之前，需要创建一个简单的用来显示游戏成绩的GUI。GUI是英文图形用户界面（Graphic User Interface）的缩写，这里指位于游戏窗口顶部用来向玩家提供信息的2D图形元素。创建GUI的步骤如下：首先在应用程序菜单处依次选中“GameObject | UI | Canvas”，如图4.25所示。关于GUI的更多信息可以在接下来的两章看到。

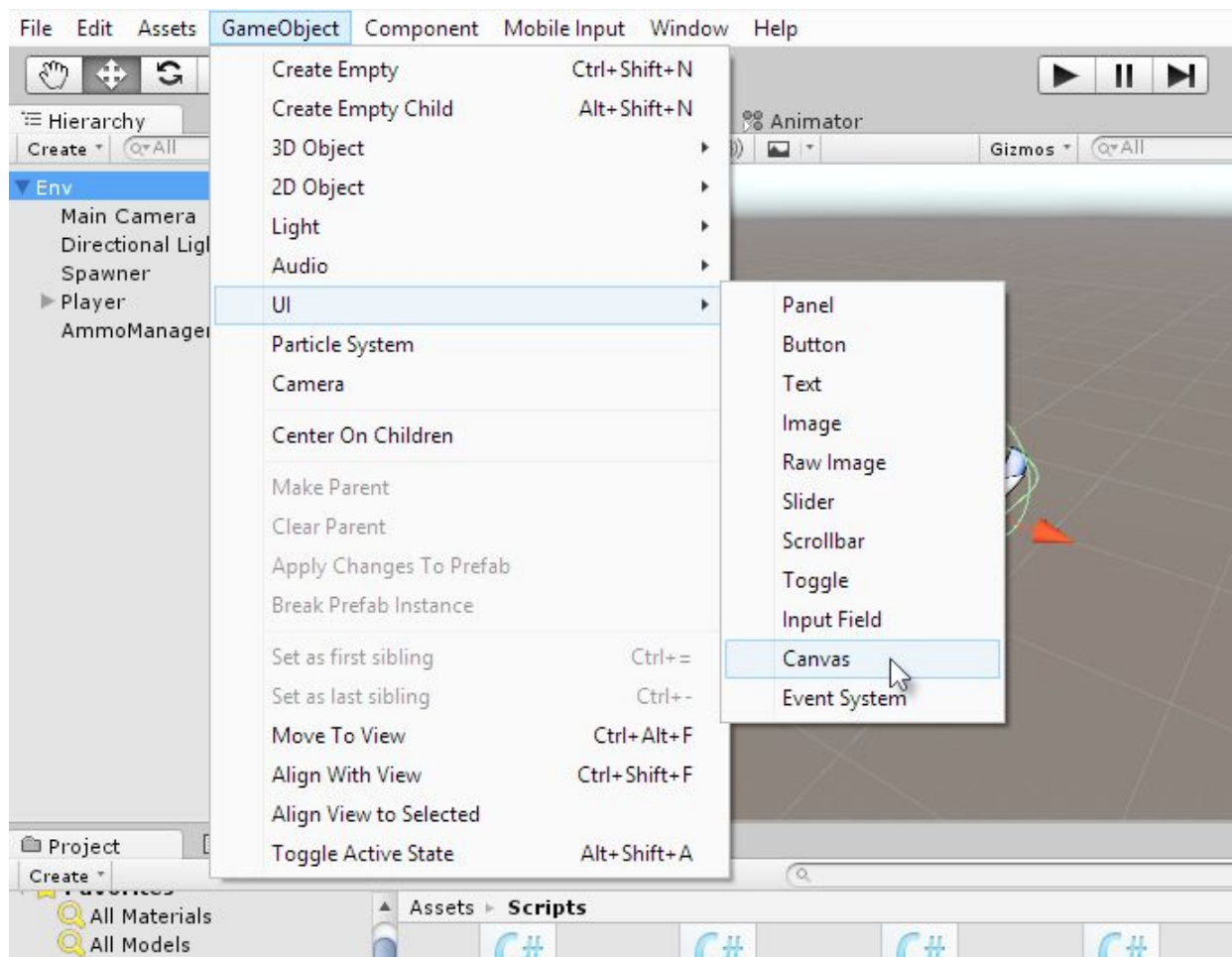


图4.25 向场景中添加一个画布（Canvas）对象

画布（Canvas）对象定义了GUI所存在的全部表面和区域，包括所有的按钮、文本和其他部件。画布对象是在场景中生成的，同样也可

以在层次（Hierarchy）面板中找到。最开始的时候，在视窗里看到的画布对象可能太大或者太小。首先在层次（Hierarchy）面板中选中画布对象，然后按下键盘上的F键以便把关注点放在画布（Canvas）对象上。这个对象显示出来是一个大的矩形，如图4.26所示。

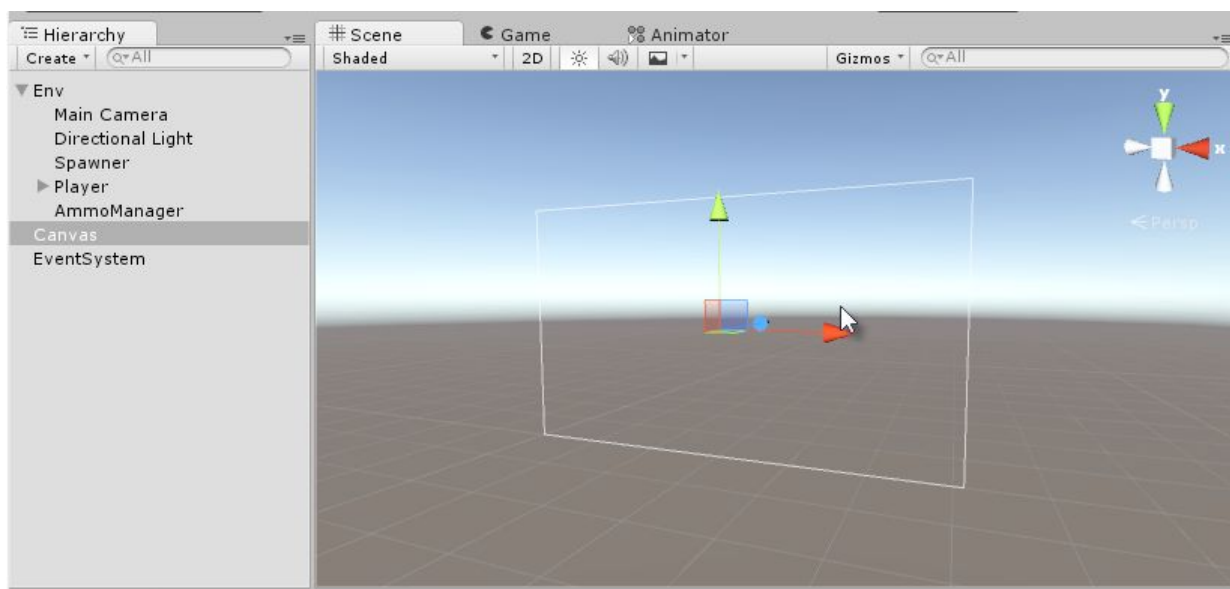


图4.26 在视窗里查看画布（Canvas）对象

在Game选项卡中并不能直接看到画布（Canvas）对象，它更像是一个容器。尽管如此，画布在很大程度上影响了它所包含对象在屏幕上的大小、位置和规模。基于这个原因，在向其中添加对象或者重新定义界面之前，最好是先重新配置画布（Canvas）对象。首先在场景选中画布（Canvas）对象，并在对象Inspector面板窗口中的“Canvas Scaler”组件处单击“UI Scale Mode”下拉列表框，然后在下拉列表框弹出的内容中选中“Scale With Screen Size”选项，并在“Reference Resolution”属性中输入一个HD分辨率，也就是说在X区域输入1920，在Y区域输入1080，如图4.27所示。

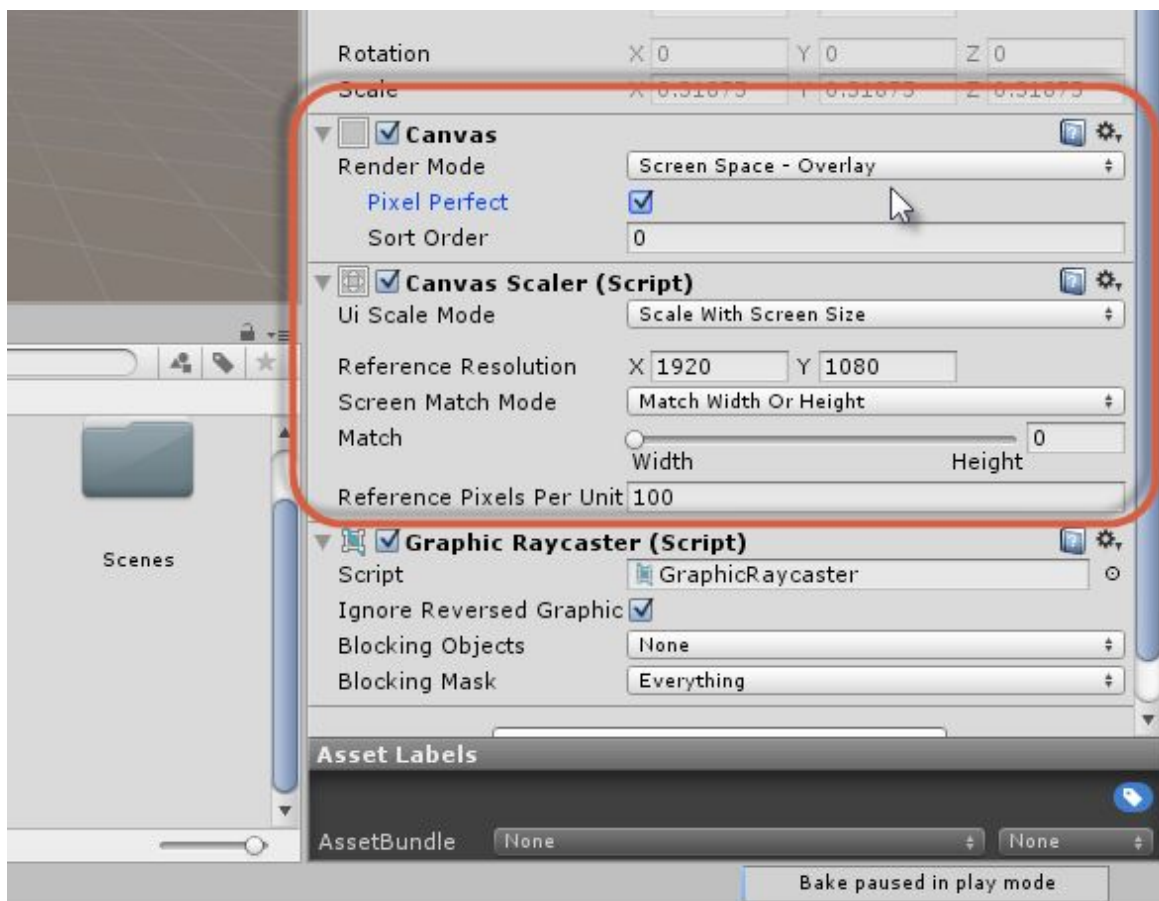


图4.27 修改“Canvas Scaler”组件的值

在将“Canvas Scaler”组件的值修改为“Scale With Screen Size”之后，游戏的用户界面将根据目标的分辨率自动拉伸或者收缩（向上或者向下），确保每个元素缩放到相同的比例，以保持整体的外观和视觉效果。这是一个创建UI的快速简单的办法，并且这种方法可以通过调整大小来适应任何分辨率。但是，如果想要保持最高质量的图像画质，这就不是一个最好的解决方案了，不过在大多数情况下，这种方法还是很适用的。还有，在进行UI设计之前，最好将场景（Scene）选项卡和游戏（Game）选项卡同时显示，当然如果配置双显示器就更好了，可以一个显示器显示场景（Scene）选项卡中的内容，另一个显示器显示游戏（Game）选项卡的内容。这样就可以在场景（Scene）选项卡中



建立用户界面，然后在游戏（Game）选项卡处对其进行预览。可以将场景（Scene）选项卡旁边的游戏（Game）选项卡拖曳出来，然后重新并排的排列，如图4.28所示。

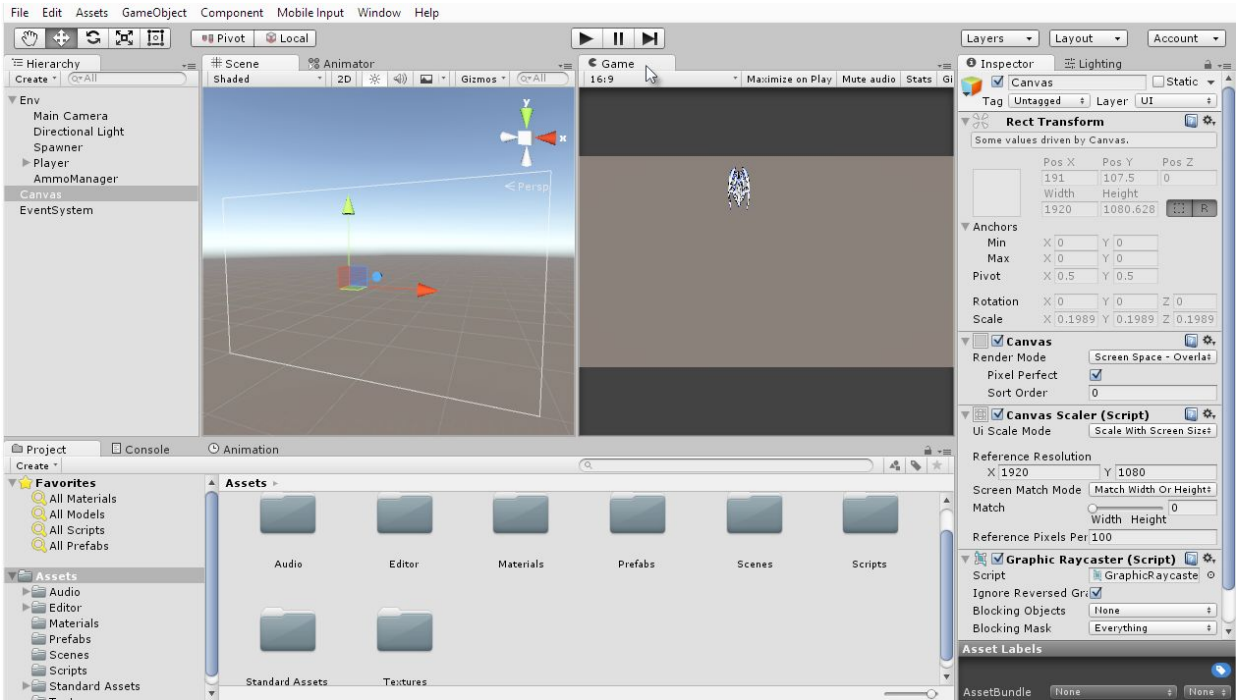


图4.28 将场景（Scene）选项卡和 游戏（Game）选项卡并排显示

现在向GUI中添加一个文本部件来显示游戏的分数，添加的步骤如下：首先在层次（Hierarchy）面板处选中画布对象，然后在这个对象上单击鼠标右键，这时会显示一个上下文菜单，然后依次选中“UI | Text”。这样就会创建一个新的文本对象，它作为画布的子对象，如图4.29所示。使用文本（Text）对象，就可以轻松地使用特定颜色、尺寸、字体来在屏幕上显示信息。

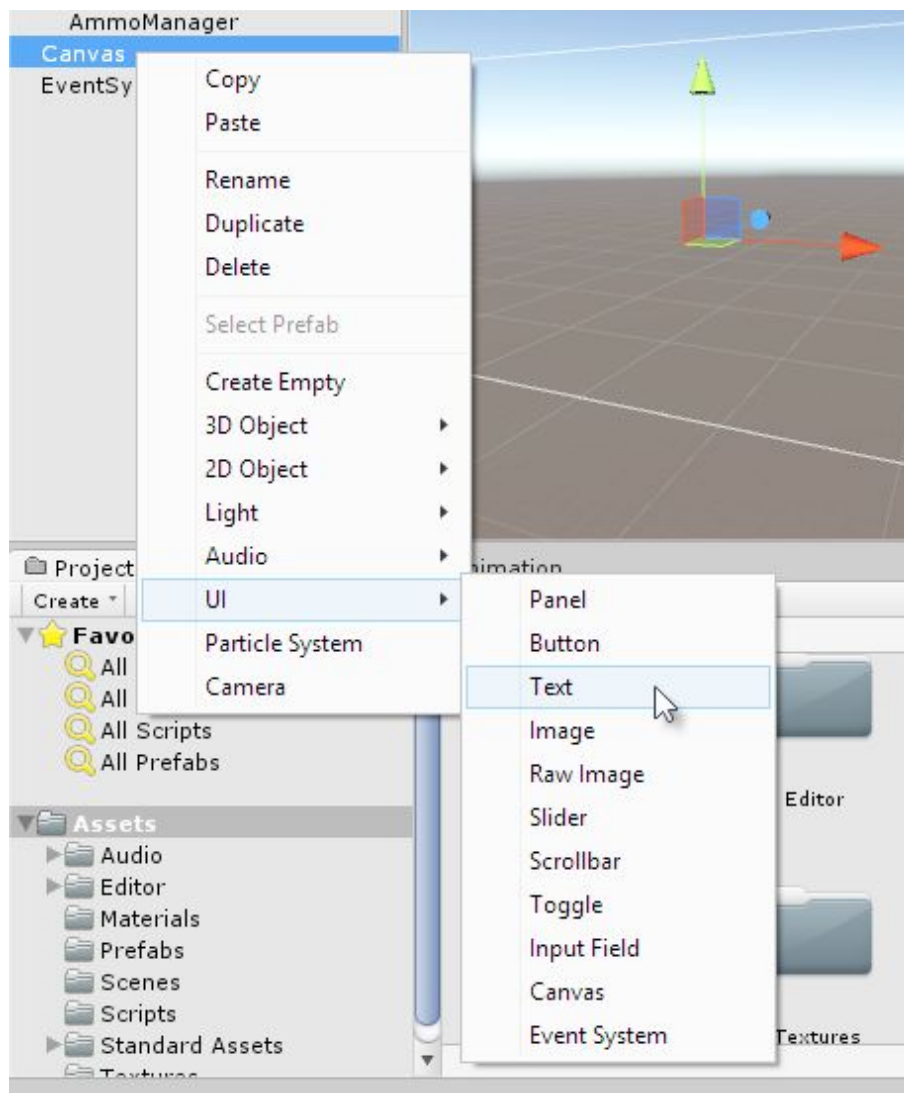


图4.29 为UI创建一个文本（Text）对象

默认情况下，文本对象无论是在场景中还是在视图，都是不可见的，虽然这个对象在层次（**Hierarchy**）面板已经列出来了。不过，当在场景中靠近一些看时，也可能看到一些很小的黑色文字。如图4.30所示，在画布和游戏（**Game**）选项卡中都可以看到这些文字。默认情况下，新的文本对象中包含一段字体很小的黑色文字。对于这个项目，需要对此进行一些修改。

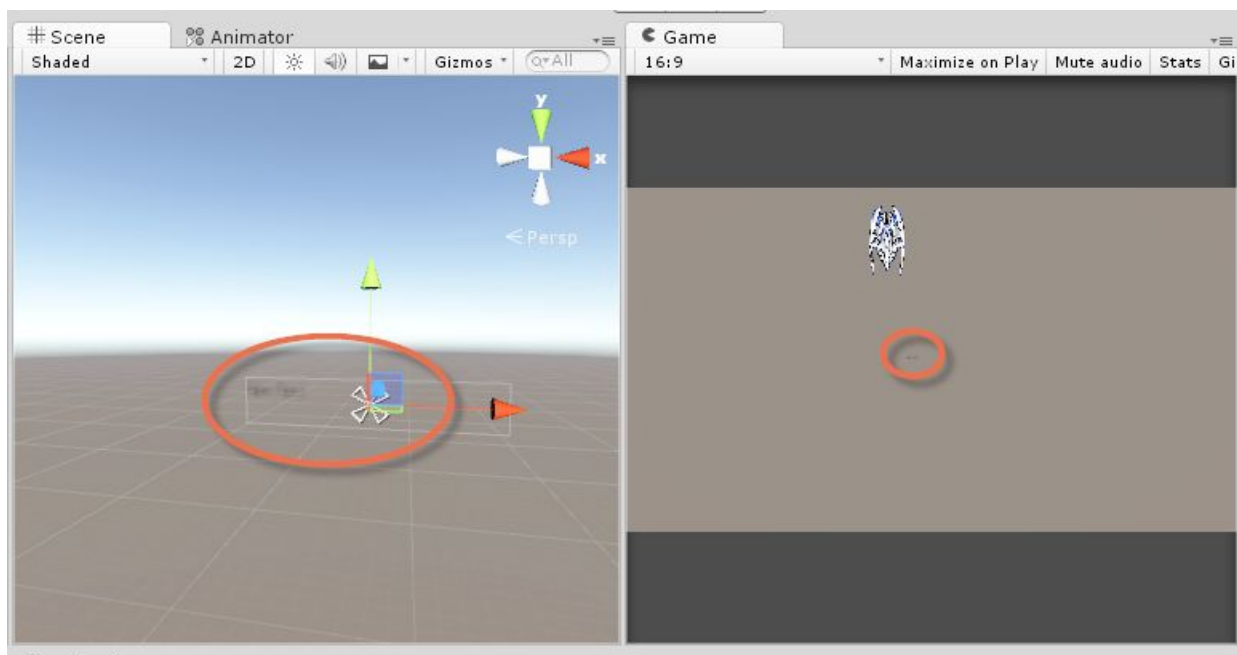


图4.30 新创建的文本对象在大多数的时候很难看清

在层次（Hierarchy）面板上选中文本对象，然后在检查（Inspector）面板中的文本组件部分，将文本的“Color”属性修改为“White”，将“Font Size”的值修改为2.0，如图4.31所示。

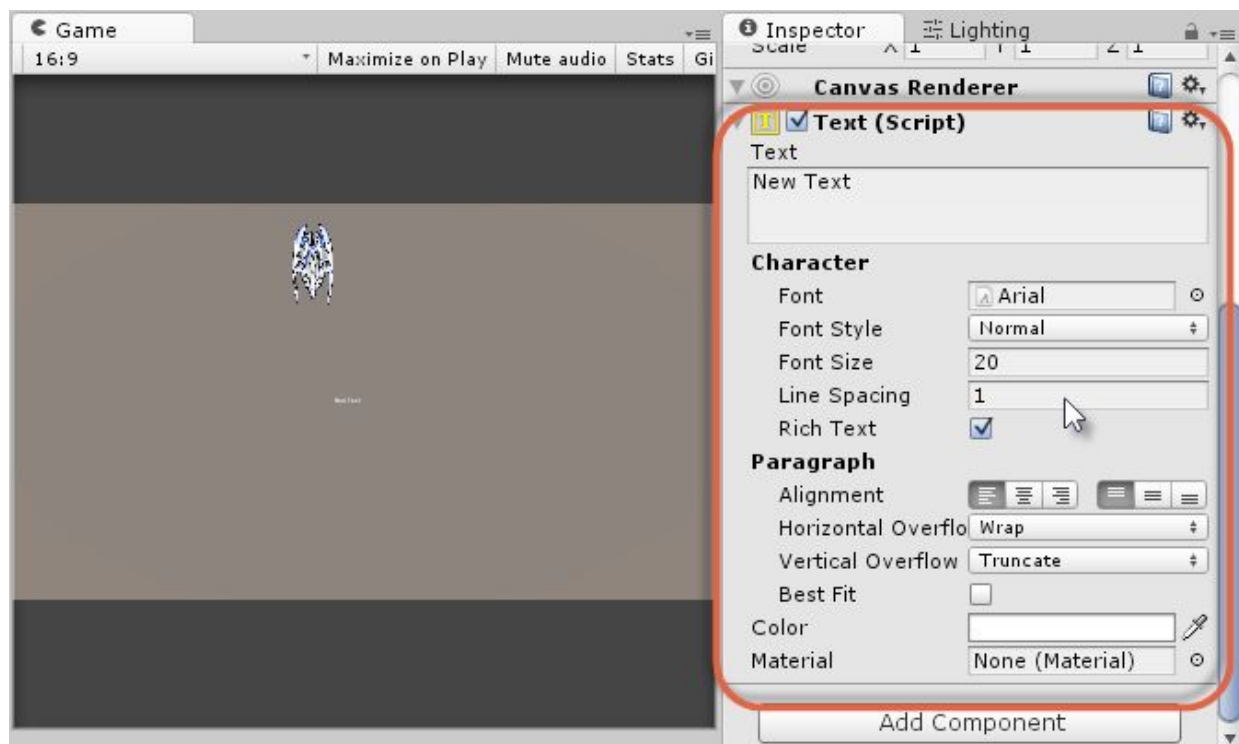


图4.31 修改文本的“Size”和“Color”

即使修改了文本的“Size”属性之后，这个对象看起来仍然很小。如果进一步增加它的“Size”值，文本可能就会从视野中消失。出现这种情况是因为每一个文本对象都有一个矩形的边界，这个边界对文本对象进行了限制。当字体的大小超出了这个边界的限制之后，文本就会自动地隐藏起来。为了解决这个问题，可以扩大文本的边界。要实现这一点，在工具栏上选中“Rect Transform”工具，如图4.32所示。

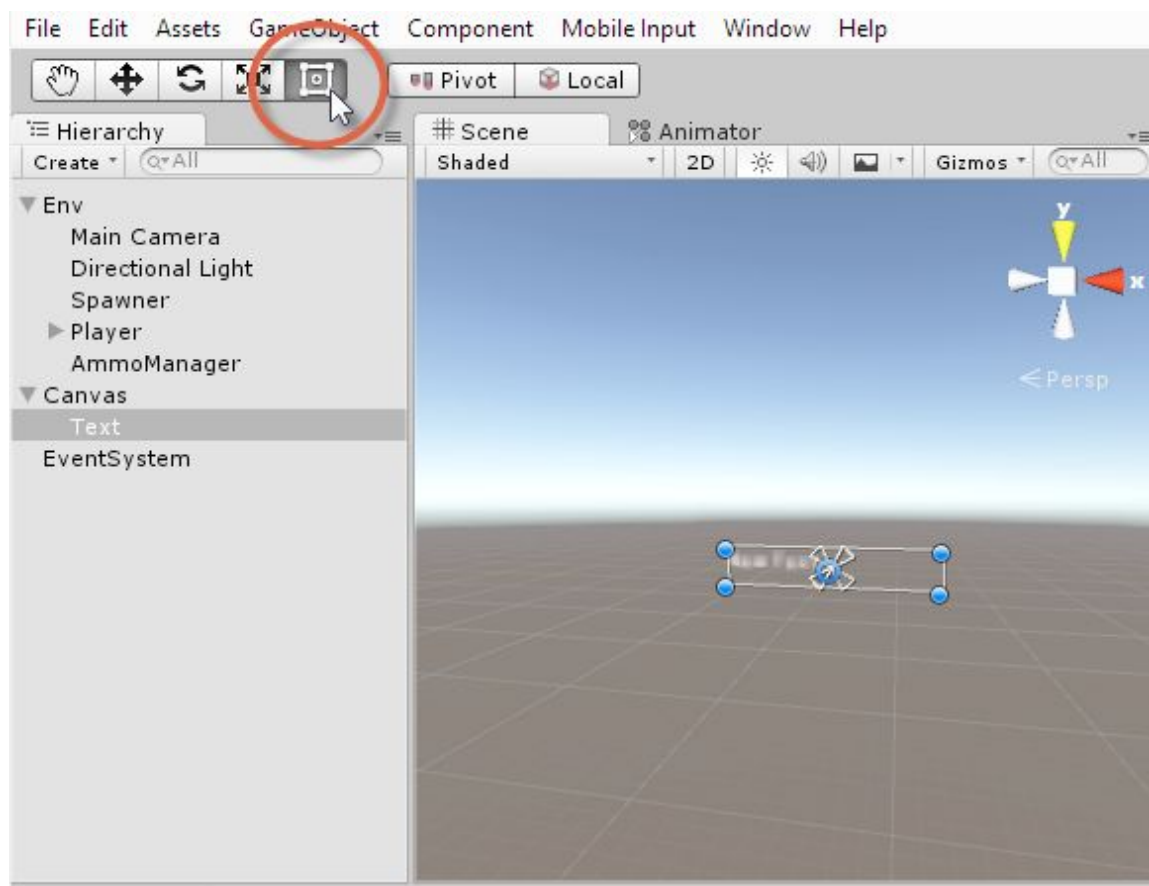


图4.32 选中“Rect Transform”工具

当激活“Rect Transform”工具之后，就可以在场景视图中查看到一个围绕在选定的文本对象外面的清晰边界，这就是矩形边界。现在来扩大这个边界的大小，以适应更大的文本，只需使用鼠标将边界拖曳到所需的位置，如图4.33所示。这样就可以扩大边界，然后将字体变大以改善文字的可读性。

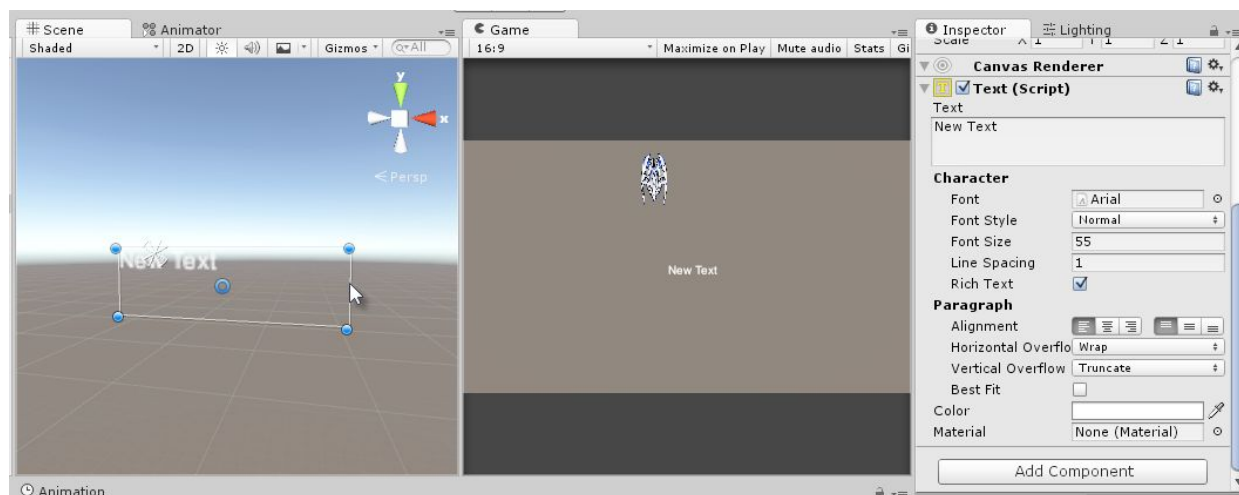


图4.33 改变矩形的大小以适应较大的字体

除了设置文本对象边界大小之外，文本也可以垂直地对齐到中心的位置。要实现这一点只需要单击垂直对齐方式中的居中按钮。对于水平对齐方式，保留默认左对齐就可以了，如图4.34所示。



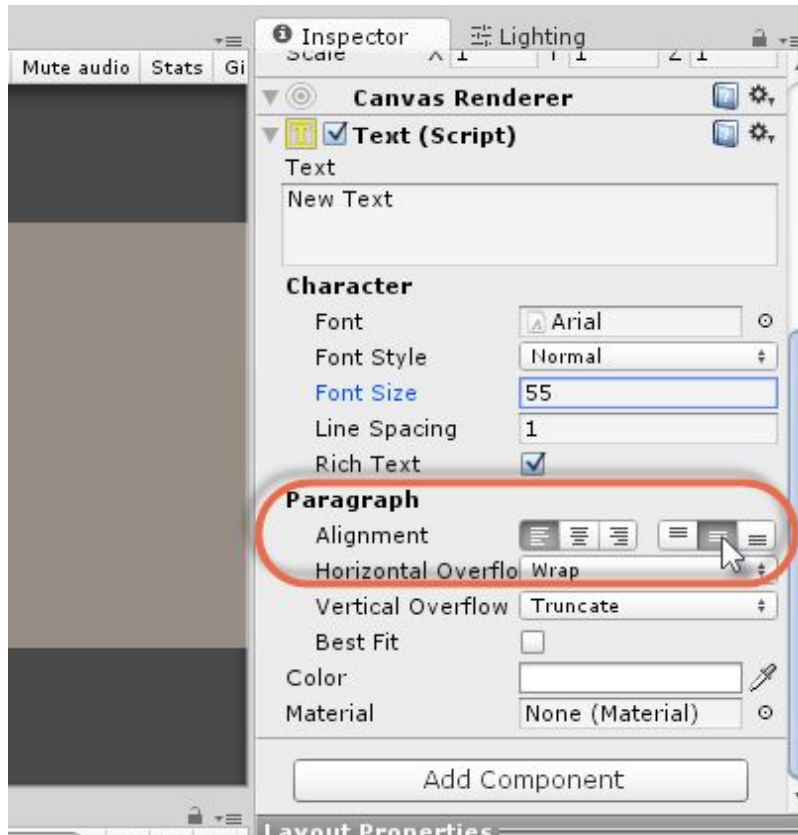


图4.34 在边界内设置文本的对齐方式

现在文本已经设置了垂直对齐的方式，还需要将整个画布容器作为一个整体来设置，即使游戏窗口的大小和位置调整了之后，这个画布容器仍然位于屏幕上相同的位置和方向。为了实现这一点，需要使用**Anchor**s。使用变形工具（**W**）将文本（**Text**）对象移动到屏幕的右上角，这个位置就是应该显示得分的位置。这个对象将会在二维的空间中移动，而不是在三维空间。当在场景视图中移动文本（**Text**）对象时，可以在游戏（**Game**）选项卡中查看它的显示是否准确，如图4.35所示。

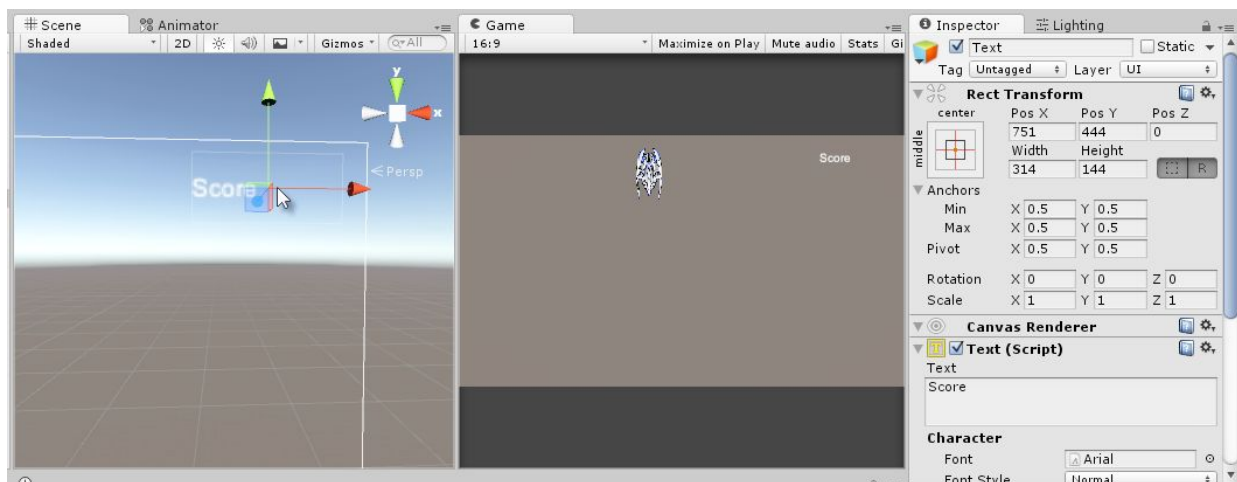


图4.35 在游戏（Game）选项卡中对Score文本（Text）对象定位

需要将文本对象固定在屏幕上（防止它移动或者变化），即使游戏选项卡被用户调整了大小。可以将对象Anchor的位置设置到屏幕的右上角，这样就可以确保文本距离Anchor的位置始终是一个固定的比例。要做到这一点，首先在对象检查（Inspector）面板的“Rect Transform”组件上单击“Anchor Presets”按钮。完成这个操作之后，就会出现一个预设的菜单，在这个菜单中可以看到一些对齐的方式。每个方式以一个小图的形式出现，每个小图上有一个红色的点代表对齐的方式，在这里选中右上对齐的方式，如图4.36所示。

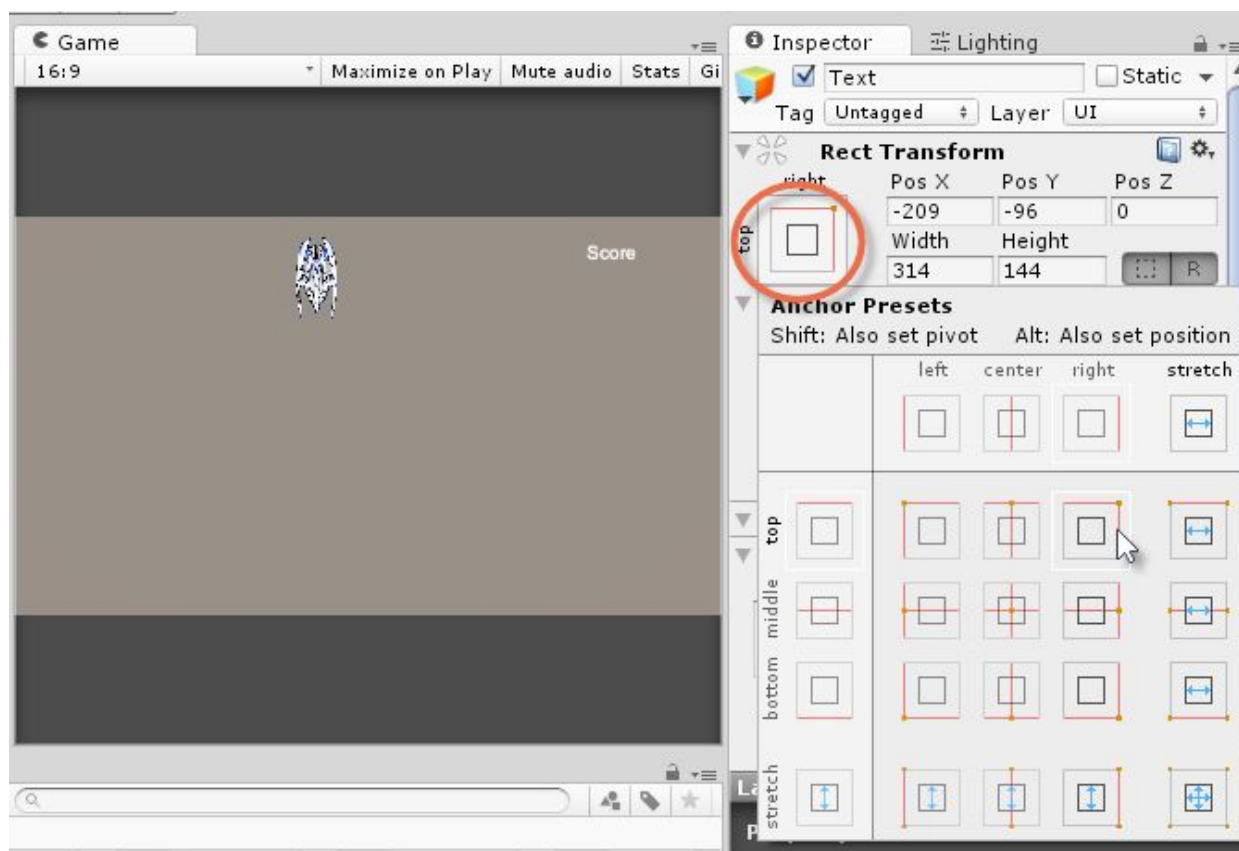


图4.36 将文本对象与屏幕对齐

现在成功创建了一个文本对象。当然，在Play模式中，这个文本的内容既不会改变，也不会显示出真实的分数。这是因为还没有向其添加实现功能的代码。不过，文本对象现在已经就位了，而且也可以随时移动它。

## 4.6 计分功能——为文本对象编写脚本

为了能在GUI中显示出玩家的得分，需要实现评分系统，也就是编写一段具有评分功能的代码。评分系统将会被添加到游戏中一个负责总体功能的GameController类中，这个类将负责游戏中所有逻辑和功

能。代码示例4.4给出了GameController类和评分系统的代码，这段代码应该添加到项目的Script文件夹中。

#### 代码示例4.4:

```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;
//-----
public class GameController : MonoBehaviour
{
    //游戏的得分
    public static int Score;

    //前缀
    public string ScorePrefix = string.Empty;

    //分数文本对象
    public Text ScoreText = null;

    //游戏结束文本
    public Text GameOverText = null;

    public static GameController ThisInstance = null;
    //-----
    void Awake()
    {
        ThisInstance = this;
    }
    //-----
    void Update()
    {
        //更新分数文本
        if(ScoreText!=null)
            ScoreText.text = ScorePrefix + Score.ToString();
    }
    //-----
    public static void GameOver()
    {
        if(ThisInstance.GameOverText!=null)
            ThisInstance.GameOverText.gameObject.SetActive(true);
    }
    //-----
}
```

下面对代码示例4.4进行总结。

- 类GameController使用了UnityEngine.ui命名空间，这一点是十分重要的，因为这个命名空间中包含了Unity中所有的UI类和对象。如果没有引入这个命名空间，那么在脚本中将无法使用任何的UI对象。
- GameController类包含了两个Text公共成员，ScoreText 和GameOverText。这两个公共成员都是Text对象，两者都是可选的，即使它们都是空的，Game Controller代码也能实现正常功能。ScoreText是一个用来显示分数文本的text GUI对象的引用。GameOverText是当触发了游戏结束条件时，用来显示任何信息的对象。

为了能使用GameController代码，需要在场景中创建一个新的空对象，将这个对象命名为“GameController”。然后将GameController脚本拖曳到这个对象上。添加之后，再将ScoreText对象拖曳到GameController的对象检查（Inspector）面板中的“Score Text”属性中，如图4.37所示。在“Score Prefix”属性中输入得分的前缀词语。得分指的是一个数字（例如1000），前缀词语指的是添加在这个得分前面的修饰性词语，这个词语可以让玩家清楚地了解这个数字的含义。

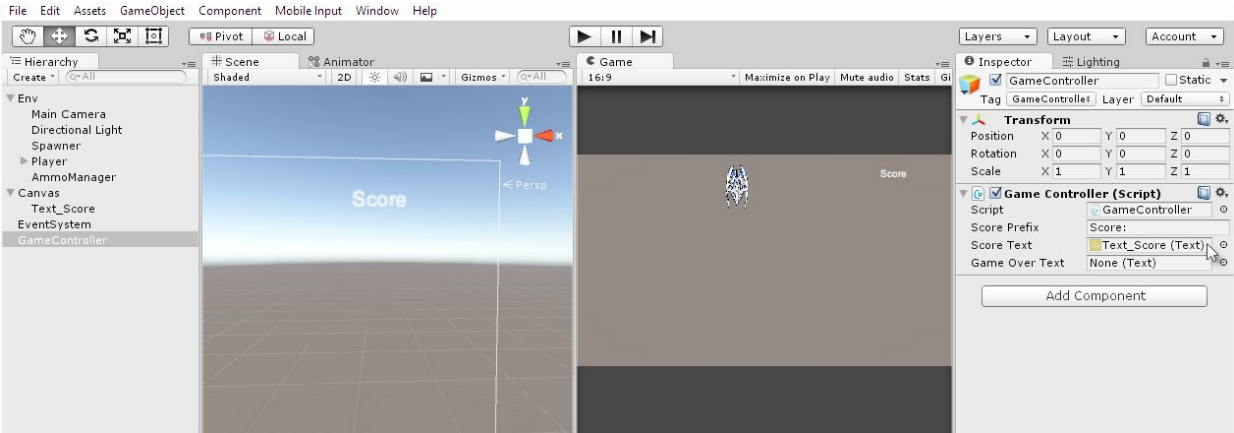


图4.37 创建一个GameController来记录游戏的得分

现在来测试这个游戏，在游戏中玩家的得分可以在游戏（Game）选项卡中的右上角看到，但是玩家的得分始终为0，一直都不会发生改变。这是因为还没有编写改变得分的代码。对于游戏来说，每当一个敌人（Enemy）对象被破坏的时候，玩家的得分都应该增加。为了实现这个功能，需创建一个新的脚本文件，命名为“ScoreOnDestroy”。这个脚本文件的内容如代码示例4.5所示。

#### 代码示例4.5:

```
using UnityEngine;
using System.Collections;
//-----
public class ScoreOnDestroy : MonoBehaviour
{
    //-----
    public int ScoreValue = 50;
    //-----
    void OnDestroy()
    {
        GameController.Score += ScoreValue;
    }
    //-----
}
//-----
```



这个脚本应该关联到所有当被摧毁时能增加点数的对象，例如敌人的身上。总的点数由ScoreValue指定。接下来，将这个脚本附加到敌人（Enemy）预设体上，首先在项目（Project）面板上选中Prefabs，然后在对象检查（Inspector）面板上，单击“Add Component”按钮，在Search文本框中输入“ScoreOnDestroy”来向预设体中添加一个组件。成功添加之后，指定要摧毁敌人的总点数。对于这个游戏，分配的总点数设置为50，如图4.38所示。

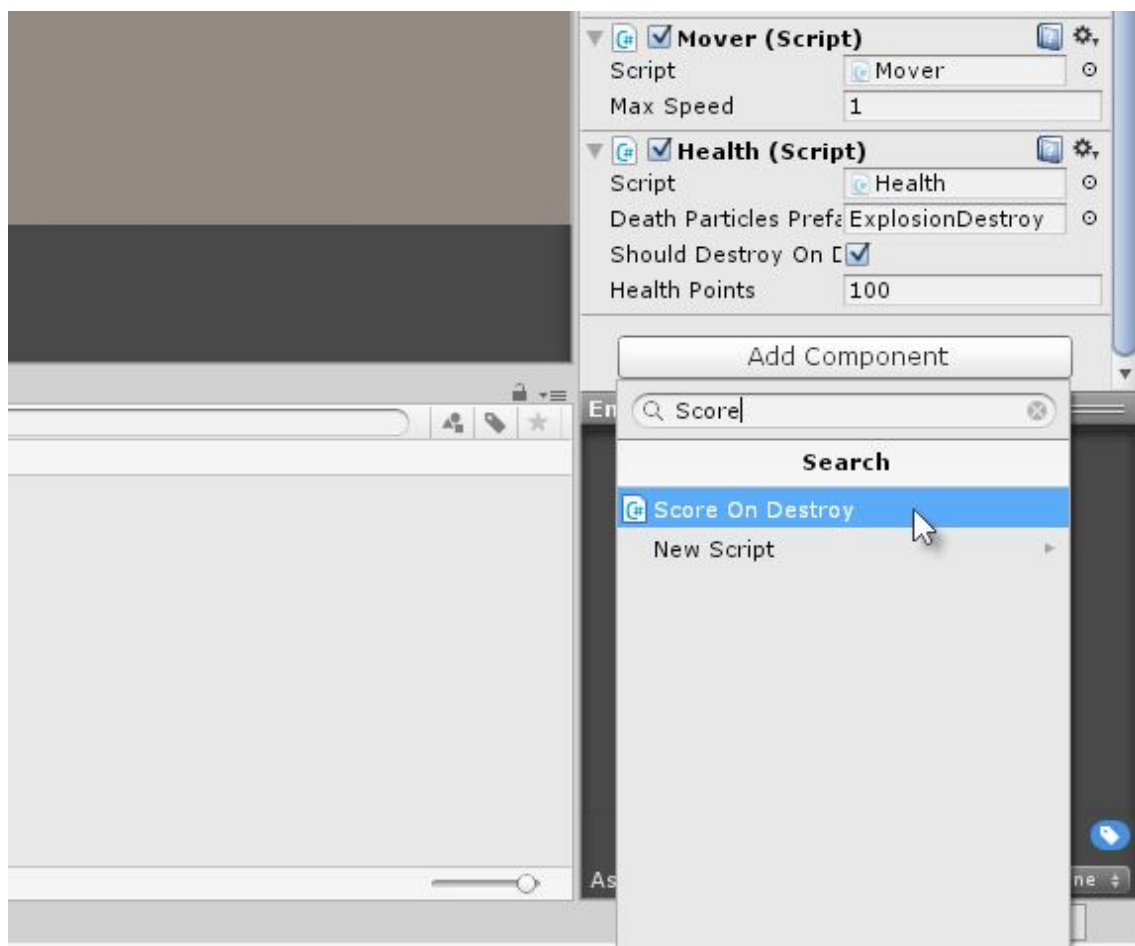


图4.38 向Enemy预设体添加一个得分组件

现在已经可以通过消灭敌人来获得分数了。对游戏进行扩展，设立高分奖励或者得分排行榜，此时游戏设计已经接近尾声了，开始准备构建游戏。接下来，对这个游戏添加一些最后的功能。

## 4.7 游戏的润色

这一节向游戏添加最后的功能。首先要做的是改善游戏的背景。到目前为止，游戏中的背景仍然是摄像机中关联的背景色。不过，既然游戏设定是发生在太空中，那么也应该为游戏添加一个太空图片作为背景。为了实现这个功能，需要在场景中创建一个新的四边形（Quad）对象来展示一张太空背景图片。在菜单栏上依次选中“GameObject | 3D Object | Quad”。转动这个对象，并将其向下移动，使它成为一个平的、垂直对齐的背景，可以调整对象的大小，如图4.39所示。

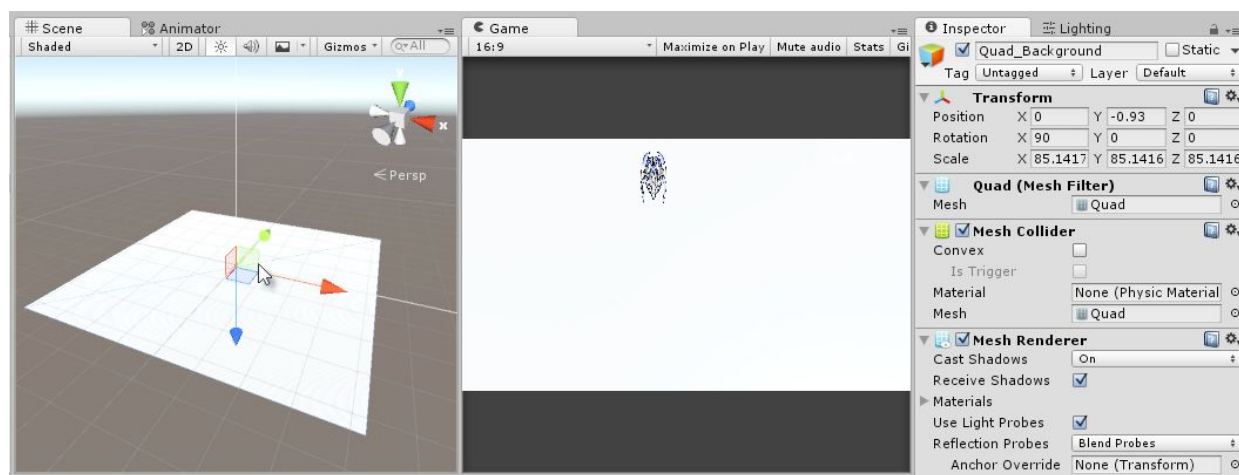


图4.39 为关卡创建一个背景，建立一个四边形（Quad）对象

将太空贴图从项目（Project）面板处拖曳到场景中的四边形对象上，将这个贴图作为四边形对象的材质。设置成功之后，选中四边形

对象然后在对象Inspector面板修改材质属性中的“Tiling”的值，将“Tiling”的“X”和“Y”的值都修改为图4.40所示的3。

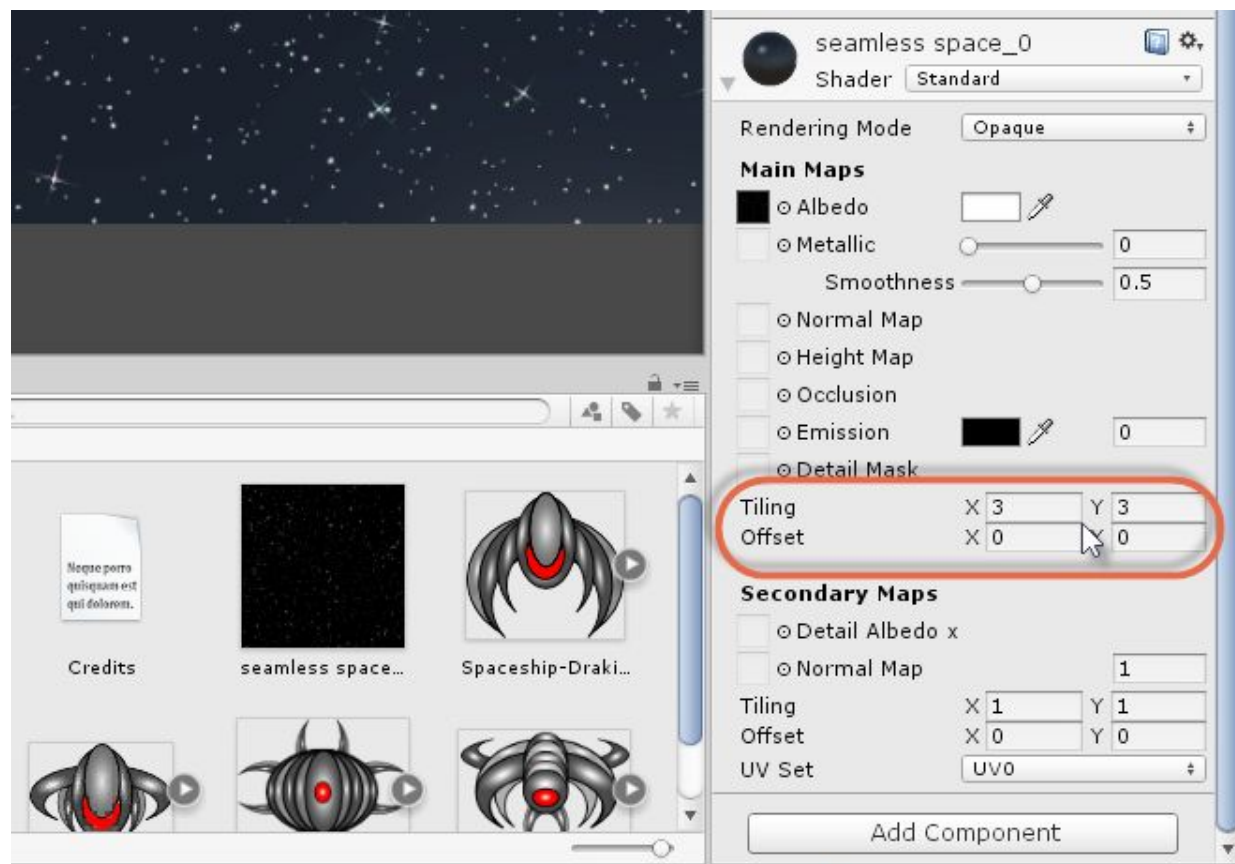


图4.40 配置贴图的tiling

如果贴图看起来是不完整的，那么就需要检查一下贴图的导入设置。具体的步骤是：在项目（Project）面板中选中贴图，然后在对象Inspector面板里，检查“Texture Type”是否被正确地设置为了“Texture”，以及“Wrap Mode”的值是否被正确地设置为了“Repeat”，如图4.41所示。

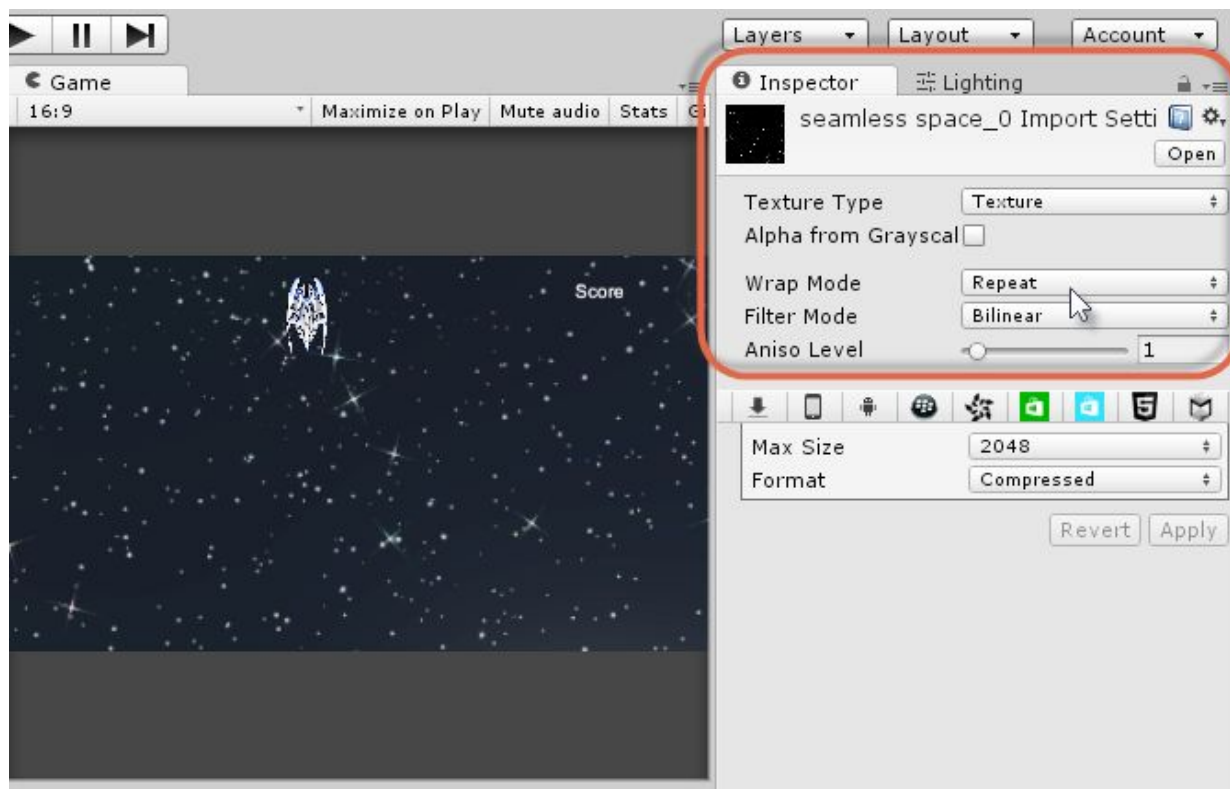


图4.41 配置贴图以实现无缝拼接

当前已经有了一个合适的背景。接下来，为这个游戏添加一些循环播放的背景音乐。首先在项目（Project）面板中的Audio文件夹中选中音乐文件，完成选中操作之后，要在对象检查（Inspector）面板确保音乐的“Load Type”属性的值为“Streaming”，并且禁用了“Preload Audio Data”属性，如图4.42所示。这样设置之后就可以减少Unity游戏开始时载入音乐的时间，无需在场景开始的时候将全部的音乐文件一次性载入到内存中。

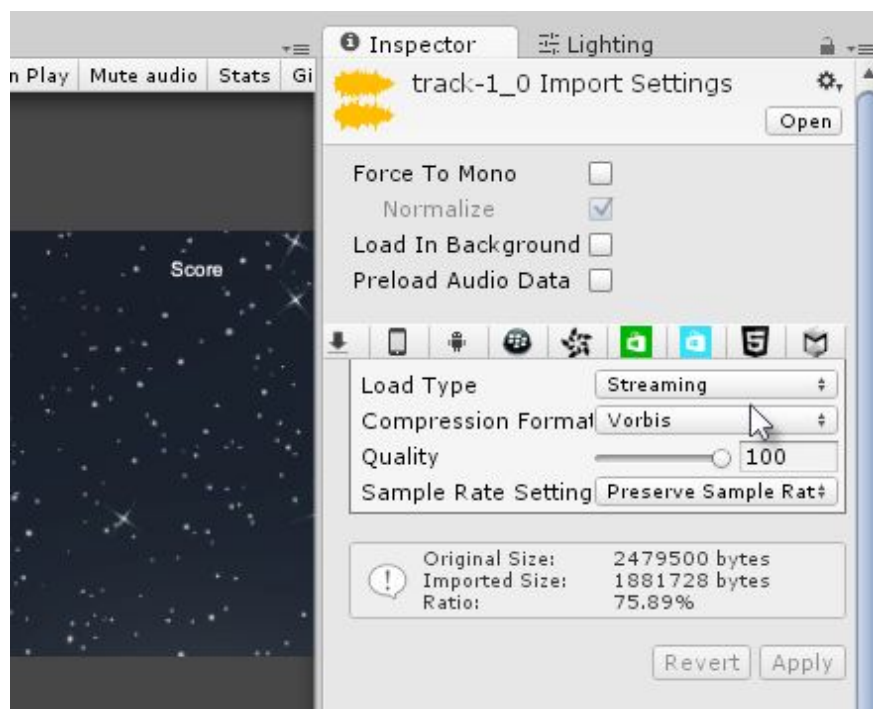


图4.42 为循环播放设置Audio文件

接下来，在场景中创建一个新的空游戏对象，并将其命名为“Music”，然后从项目（Project）面板上将音频文件拖动到Music对象上，并作为这个对象的一个“Audio Source”组件。“Audio Source”组件用来播放音效和音乐，如图4.43所示。

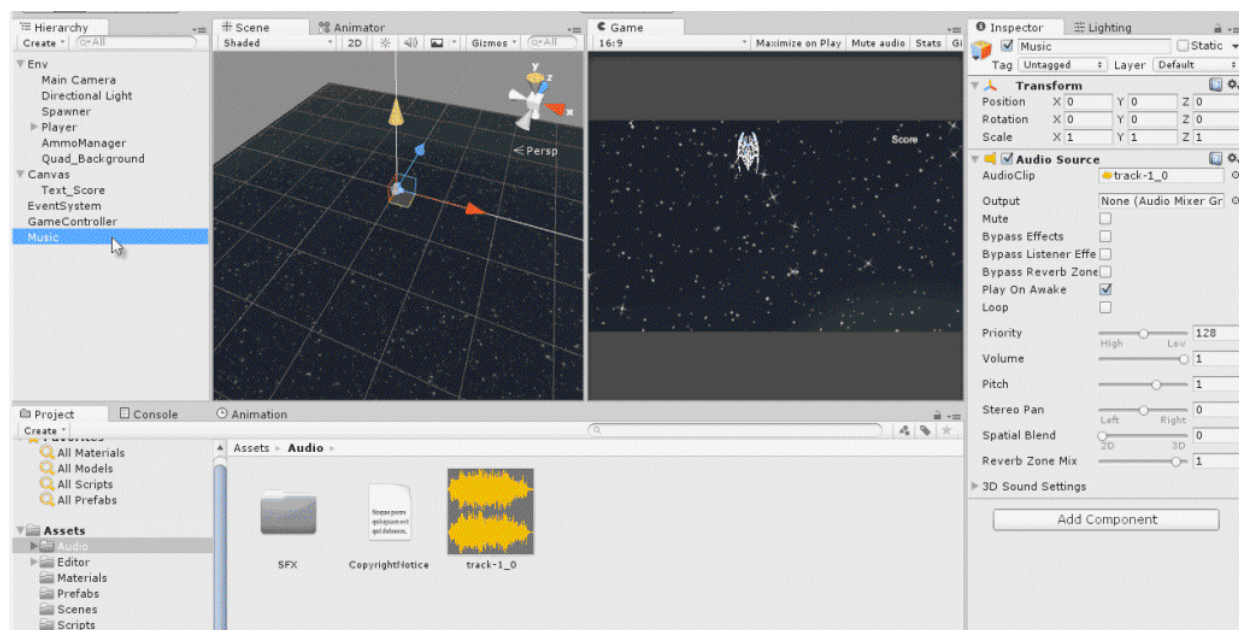


图4.43 创建一个具有AudioSource 组件的游戏对象

在对象检查（Inspector）面板中的“Audio Source”组件处，将“Play On Awake”和“Loop”两个单选框都选中，这样就可以确保从关卡开始一直到结束的整个游戏过程中，音乐都能一直播放。另外“Spatial Blend”的值应该设置为0，这就意味着这个声音是以2D形式播放的。简而言之，2D的声音是指从始至终一直以恒定的音量播放，而跟玩家的位置无关，这是因为2D的声音是没有空间定位功能的。而3D的声音却恰恰相反，它会根据玩家所处的位置不同而不同，这种声音在游戏中主要是指枪声、脚步声、爆炸声等，如图4.44所示。



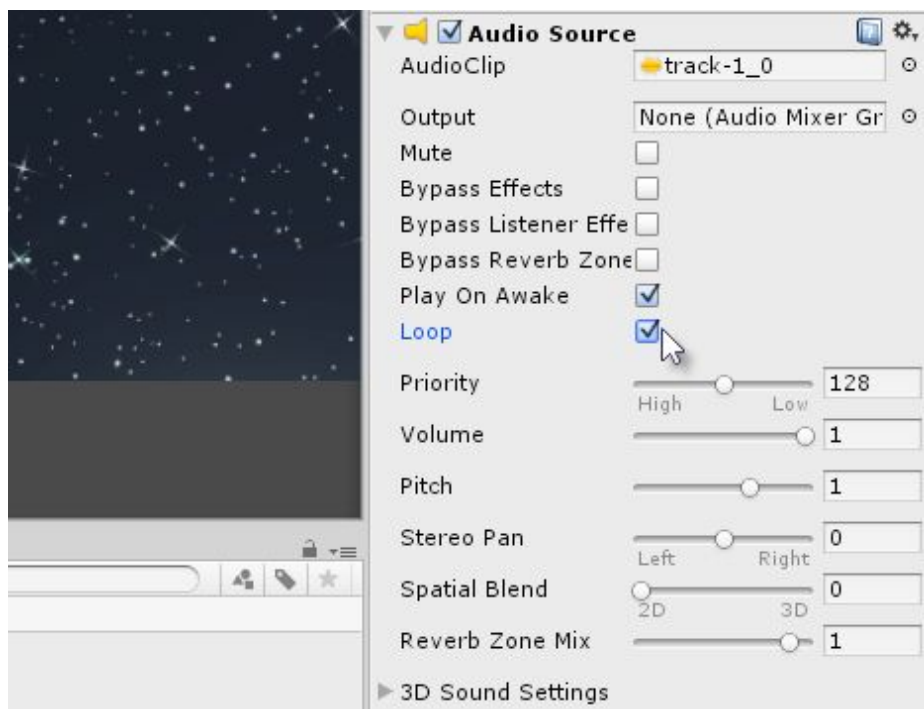


图4.44 循环播放一个音乐文件

接下来又到了游戏的测试时间了！在工具栏上单击“Play”按钮，然后对其进行测试，如果音乐声没有正常响起来，在游戏（Game）选项卡中检查“Mute Audio”按钮是否被启用，如图4.45所示。

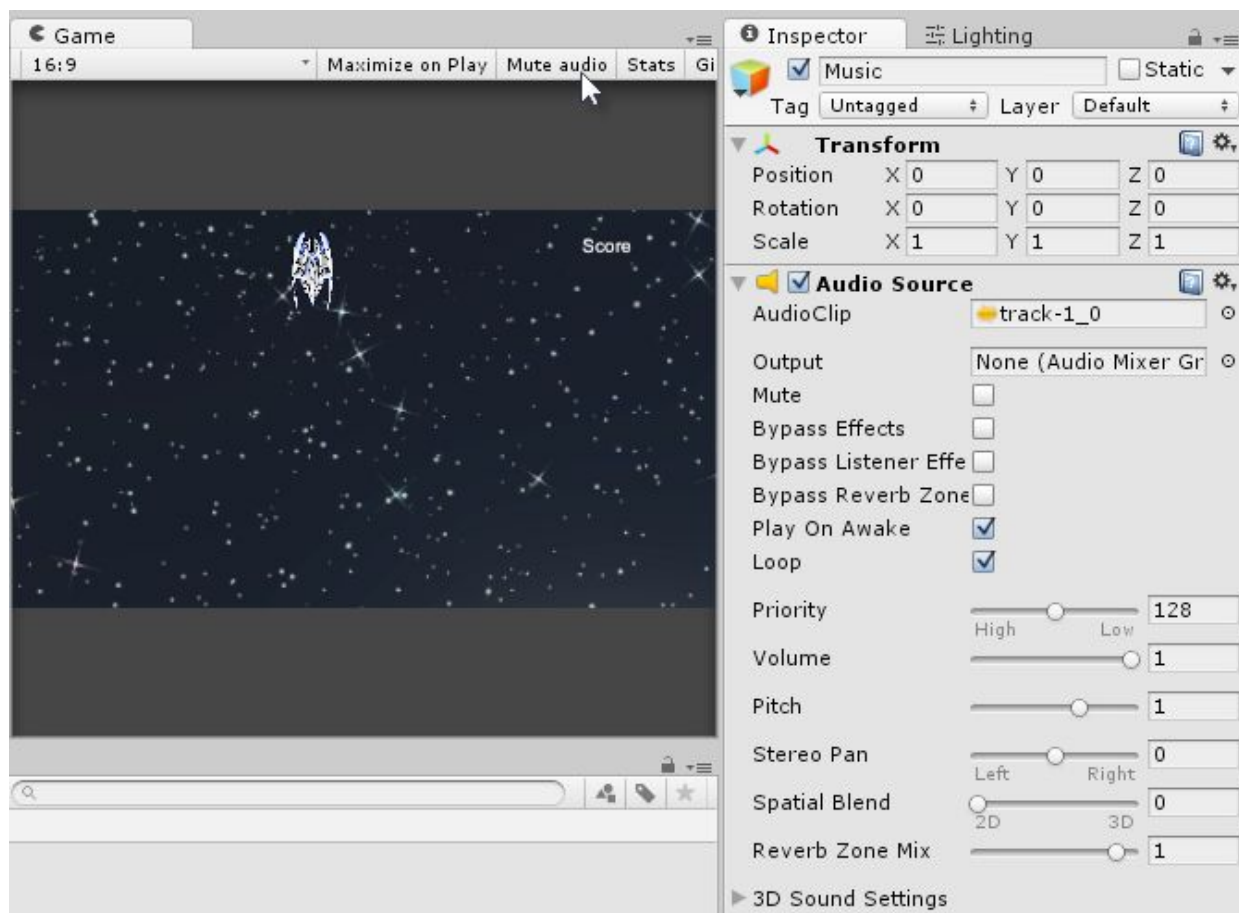


图4.45 进行游戏测试

## 4.8 测试与调试

当整个游戏开发完成之后，接下来需要做的就是花费大量的时间对游戏进行测试，以及花费大量的精力对游戏进行调试，从而消除游戏中的bug和失误。对于这个项目实例来说，只需要很少的调试和测试工作，并不是因为游戏很简单，而是本书的作者在本书的写作之前已经花费了大量的时间和经历对这个实例进行检查和测试，以确保实例的正确性，好为读者提供优秀的阅读体验。对于其他项目来说，就需要完成大量的测试工作。可以使用Stats面板来开始测试工作。在游戏

(Game) 选项卡上单击“Stats”按钮就可以打开这个面板，如图4.46所示。

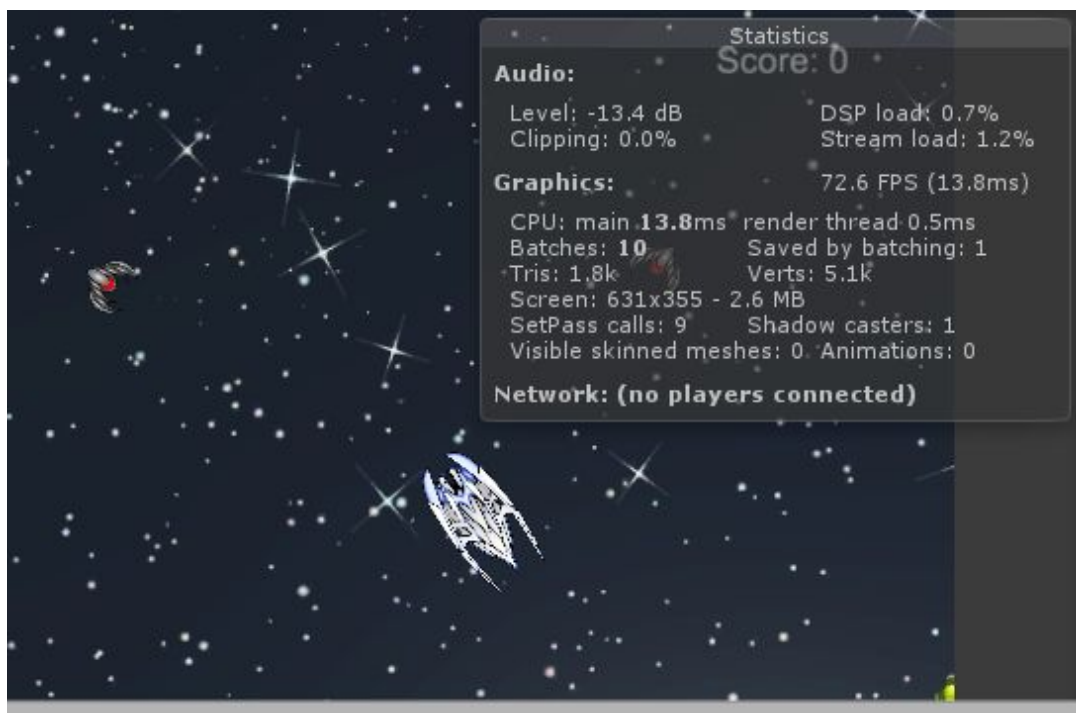


图4.46 在Stats面板上查看游戏的性能

关于Stats面板的详细信息可以在本书的第2章中找到。如果想对此深入了解，可以访问Unity的在线文档

<http://docs.Unity3d.com/Manual/RenderingStatistics.html>。

另一个调试工具就是Profiler，这个工具是十分有用的。当在Stats面板中发现了一个常见的问题后，例如低的FPS，如果希望对此进行深入研究，并找出问题的所在，就需要用到这个工具。Profiler的详细信息将在第6章中介绍，这里仅仅对它进行一个简单的介绍。可以在应用程序菜单中依次选中“Window | Profiler”来获取Profiler工具，完成这个操作之后，将会出现图4.47所示的Profiler窗口。

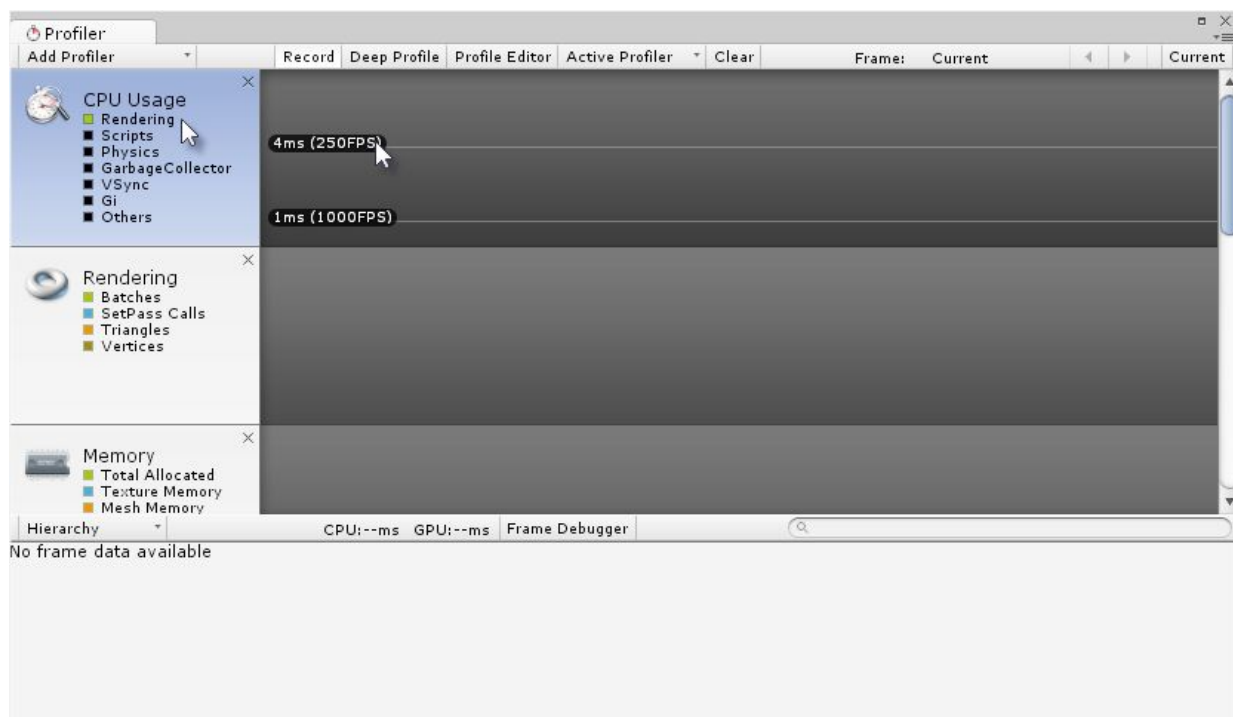


图4.47 打开Profiler窗口

当Profiler窗口打开之后，单击工具栏上的“Play”按钮来测试游戏。这时，Profiler窗口中会以彩色图的形式展示一些数据，如图4.48所示，绿色的曲线表示数据渲染的性能。要想读懂这些曲线，需要一些经验，但是有一条通用的规则：注意波峰，也就是图表中出现的剧烈波动（剧烈起伏），因为这可能意味着一个问题，特别是起伏和帧速率下降相一致的时候。

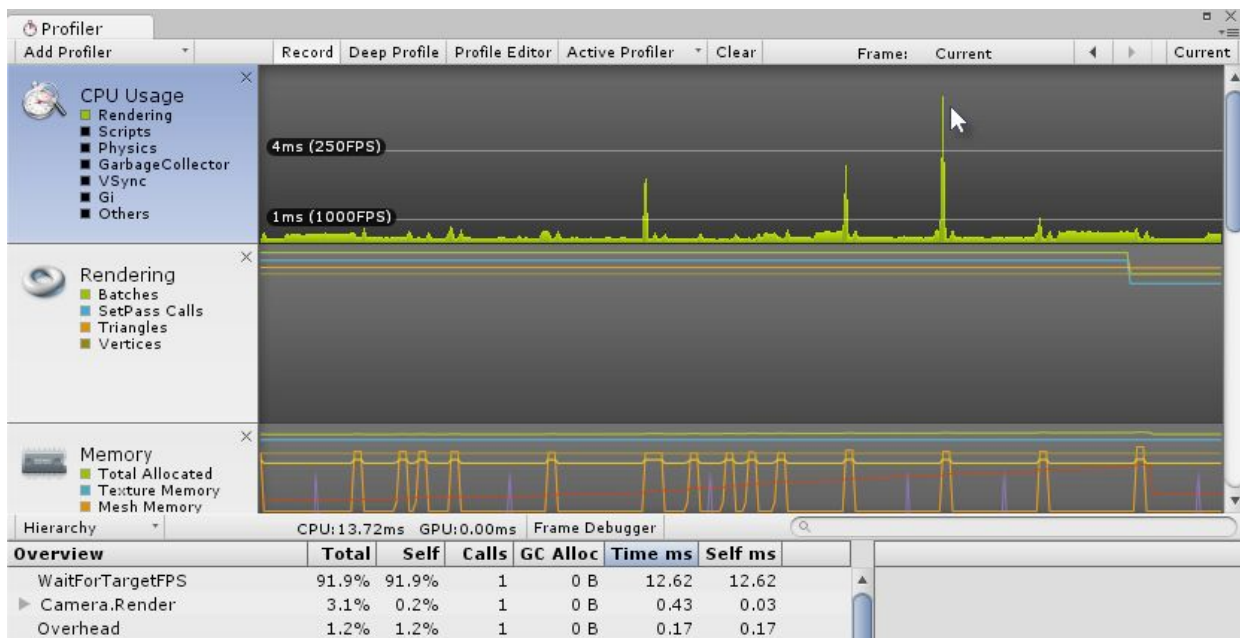


图4.48 游戏进行时的数据所填充的Profiler

如果进一步深入研究，就先暂停游戏，然后单击图形。水平轴（X轴）代表的是最新的帧，垂直轴代表的是工作负荷。当单击图中某一帧的时候，就会在帧上添加一个线性的标记，表示要对这个帧进行深入的研究。在图的下方，列出了当前帧的所有主要进程，这是一个典型的按照进程所占有的工作量排序的列表，工作量最大的排在了最上方，如图4.49所示。

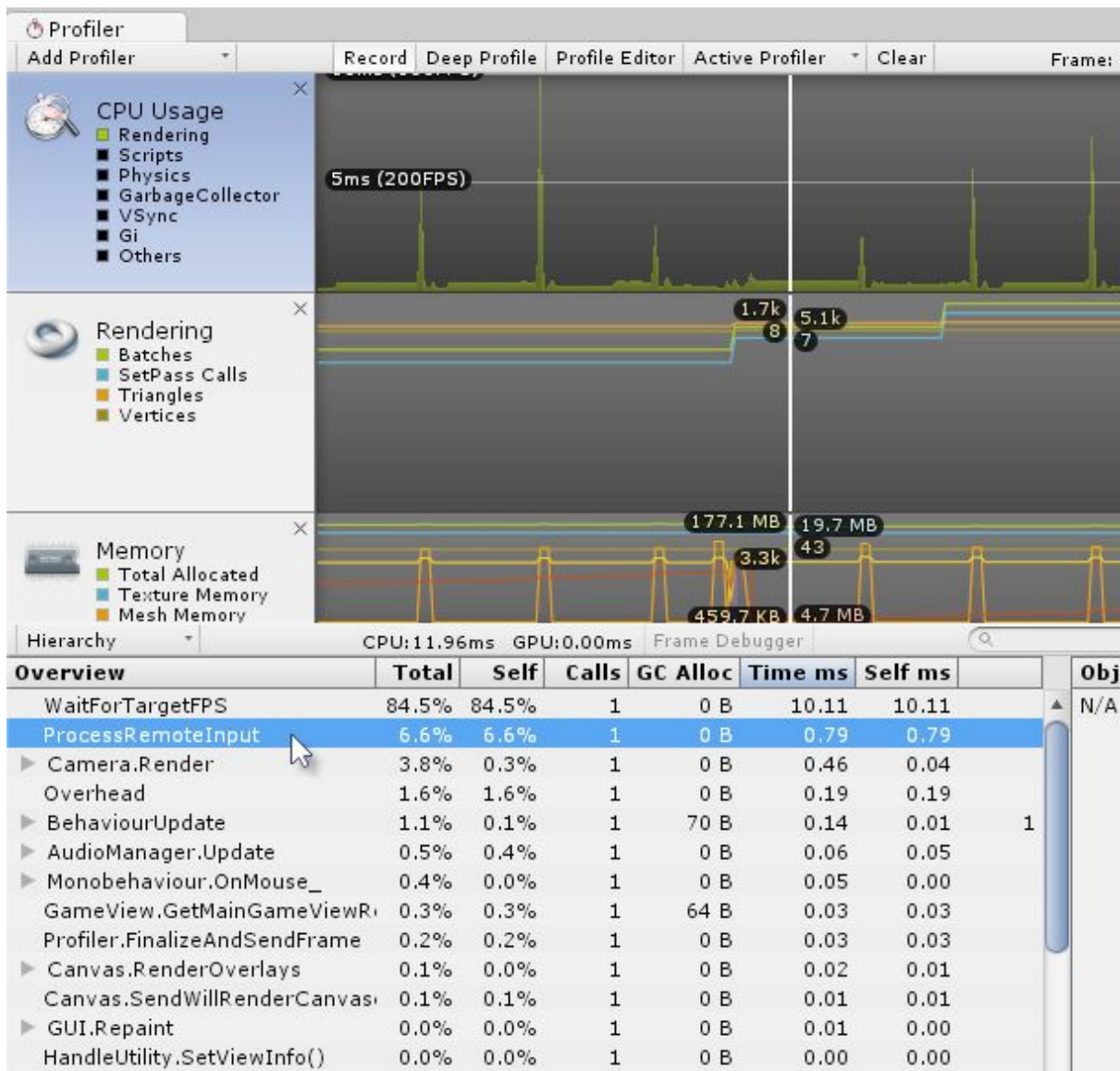


图4.49 使用Profiler研究性能数据



关于Profiler的更多详细信息可以在Unity的在线帮助文档  
<http://docs.Unity3d.com/Manual/Profiler.html>处找到。



## 4.9 游戏的构建

现在已经可以将完成的游戏打包成为可独立执行的文件发送给亲朋好友，或者测试者了！这个过程和第2章中介绍的构建过程是一样的。具体的过程是从应用程序菜单中依次选中“**File | Build Settings**”，然后在弹出的**Build**对话框中，单击“**Add Current**”按钮，将关卡添加到**Level**列表中。或者，也可以将关卡从项目（**Project**）面板上拖曳到**Level**列表上，如图4.50所示。

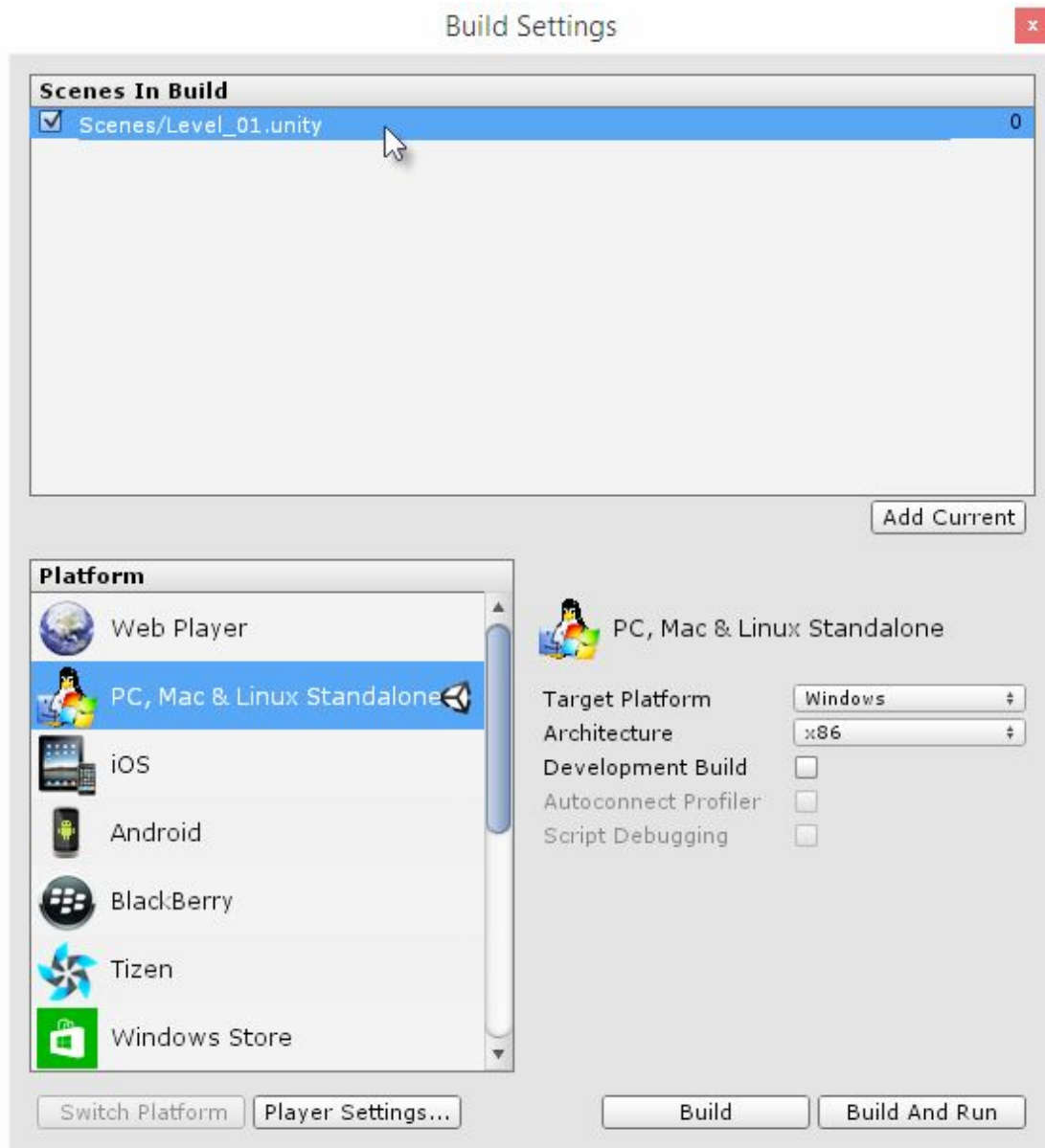


图4.50 构建（Build）宇宙设计游戏的准备

对于这个游戏，目标平台应该是Windows。因此，应该在Platform平台列表上选中“PC, Mac & Linux Standalone”选项。如果“Switch Platform”按钮（位于左下方）不是灰色的，那么需要按下这个按钮，这样Unity就会按照所选定的平台来构建（Build）游戏，而不是其他的平台。单击“Build And Run”按钮之后，Unity会要求在电脑上选择一个

用来保存构建文件的文件夹，这个文件夹用来输出和保存游戏文件。当创建成功之后，就可以双击这个可执行文件，完成对游戏文件的运行和测试，如图4.51所示。

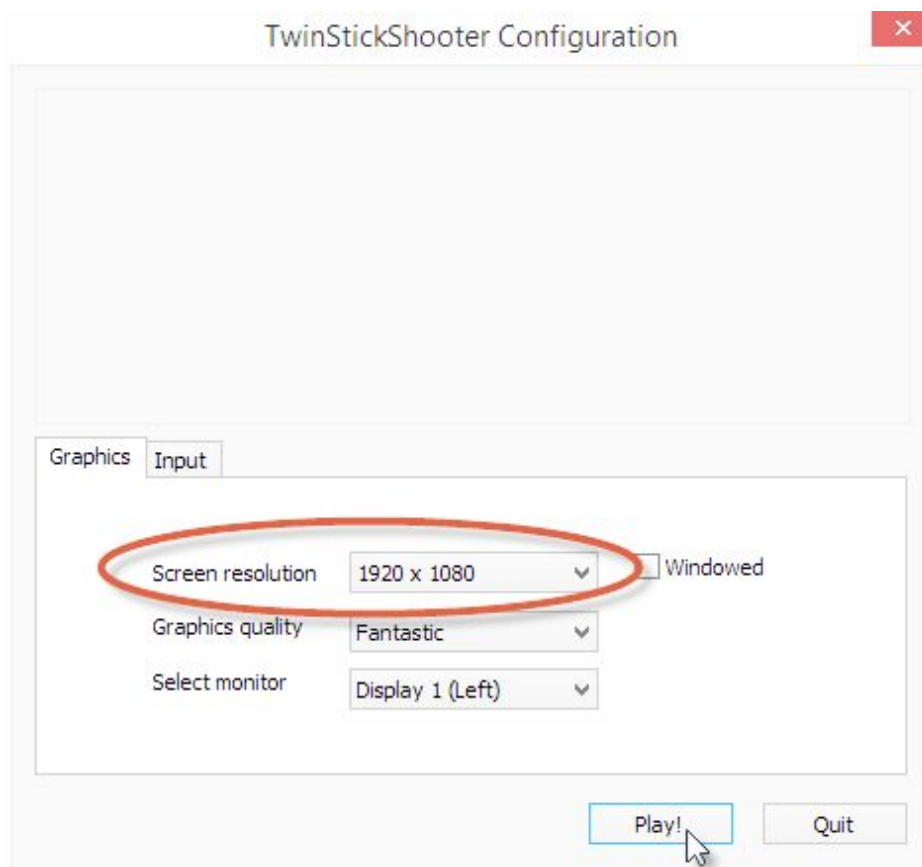


图4.51 将游戏在Windows环境下进行测试运行

## 4.10 小结

至此，已经完成了两个Unity项目。第一个项目是一个硬币采集游戏，第二个是一个双轴射击游戏。两者都是比较简单的游戏，它们既没有依赖于高级的力学机制，也没有显示出复杂的功能。不过，即使再复杂的游戏，也还是由基本功能所组成的，前面讲解过的内容中涵

盖了这些基本功能。这也就是为什么要对Unity的实例进行深入理解。接下来，将要创建更多的2D游戏，会涉及到更多的关于界面、图像精灵、物理属性等内容。

## 第5章 二维冒险游戏 (I)

本章将开始一个全新的项目，这是一个二维的冒险游戏。在这个游戏中，玩家将控制一个来自外星的角色在一个充满危险的世界完成任务。这个项目将包含以前章节中讲过的技术和思想，同样也会引入新的各种技术，例如复杂的碰撞、二维物体的物理属性、单例模式（Singleton）、静态（Statics）等。总之，本章将会涵盖如下主题：

- 二维角色和玩家的运动
- 复杂的、多个部件组成的角色
- 关卡的设计
- 二维物理属性以及碰撞的检测



本章的原始项目和资源可以在本书配套文件中的“Chapter 05/Start”文件夹中找到，如果没有建立自己的项目，就可以使用这个文件来开始本章的学习。

### 5.1 二维冒险游戏——开始

冒险游戏需要玩家充分发挥他们、智慧、敏捷、敏锐的观察能力以及技巧，才能获得胜利。这种游戏中包含了一些有危险的障碍、具有挑战性的任务，以及角色之间的互动，但这种互动并不是第一人称射击游

戏那种的全面行动。要开发的这个冒险游戏也是如此。图5.1所示为一个游戏进行中的截图。在这个游戏中，玩家将需要使用键盘上的方向键或者“W”“A”“S”“D”4个键来控制角色的移动。此外，他们还可以使用空格键来完成跳跃动作，靠近游戏中的角色来实现与他们的互动。在游戏中，玩家将从NPC角色处领取寻找一个被隐藏起来的古代宝石的任务。玩家必须突破重重险阻才能找到这个宝石，然后将这个宝石带回到NPC那里完成任务。

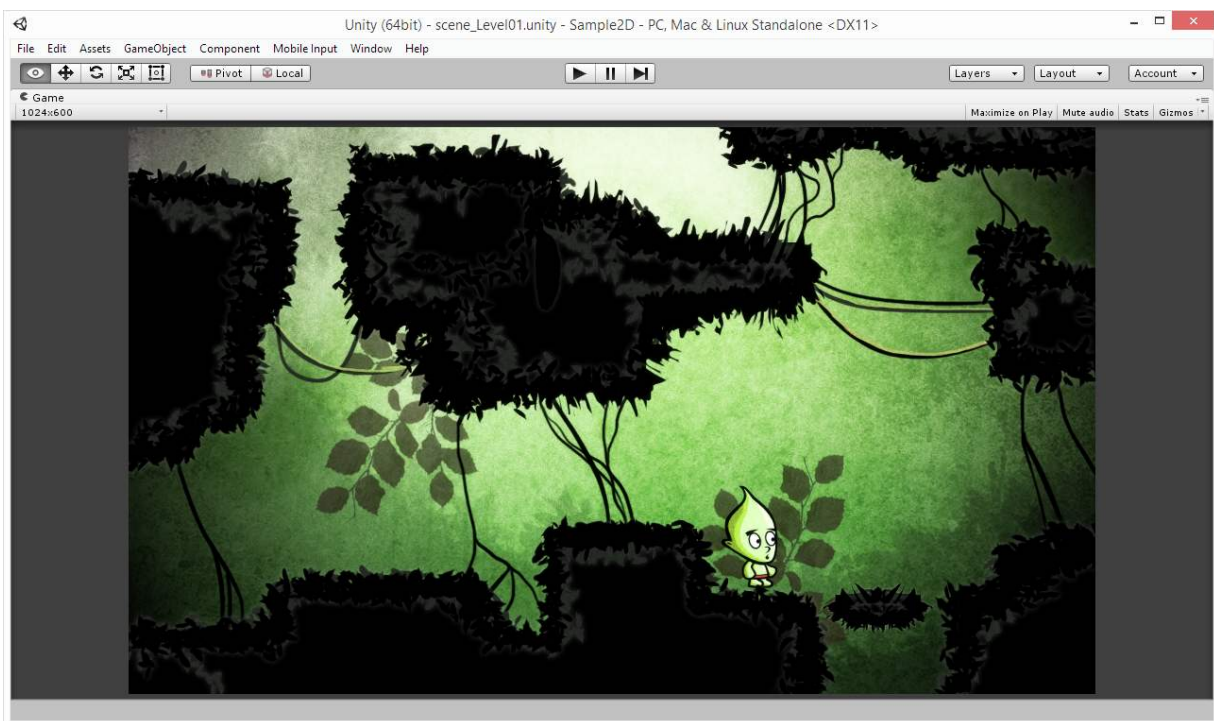


图5.1 要创建的二维冒险游戏截图

以创建一个新的空Unity项目作为游戏设计的开始，然后倒入“Particles”“Effects”“Characters”“2D”“ParticleSystems”和“CrossPlatformInput”等资源包。可以在“Project Creation Wizard”中导入这些资源，也可以通过在应用程序菜单中选中“Assets | Import Packages”实现，如图5.2所示。具体的导入过程和第1章中导入标准资源包是一样的。



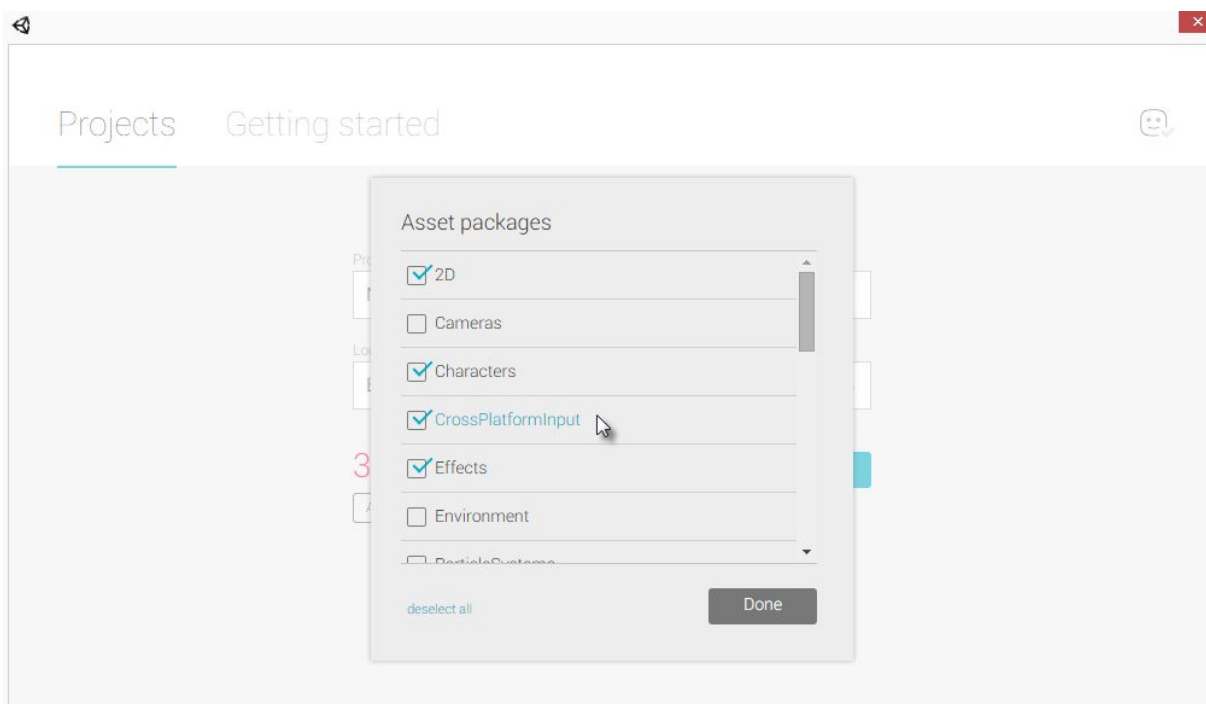


图5.2 在项目创建屏幕中为创建的新项目导入资源包

## 5.2 资源的导入

从上一节创建的空项目开始，接着导入需要使用的包括玩家角色和环境在内的贴图资源。这些需要导入的资源可以在本书的配套文件中的“Chapter05/Assets”文件夹中找到。在这里，利用Windows的资源管理器或者Mac的Finder选中所有的贴图，然后将它们拖曳到Unity项目

（Project）面板中的Textures文件夹（如果这里不存在这个文件夹，可以创建一个）。完成操作之后，就将所有的贴图导入到了当前项目中，如图5.3所示。

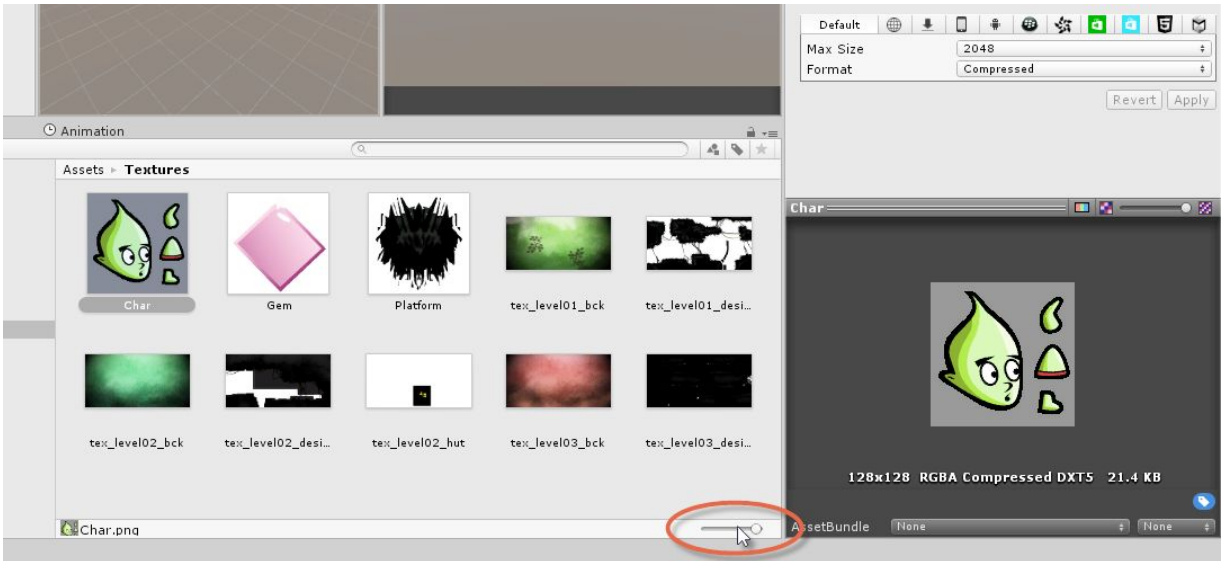


图5.3 向项目中导入贴图资源



请记住，始终都可以使用缩略图调整滑块（位于项目（Project）面板右下角）来调整缩略图大小，这样就可以对贴图资源进行观察了。

默认情况下，Unity会假定所有导入的贴图都要应用到场景中的三维模型，例如立方体、球体和网格物体上。在大多数情况下，这个设定是正确的，因为大多数的游戏都是三维的。然而，对于二维游戏来说，例如当前正在开发的这个游戏，设置应该是完全不同的。在本例中，对象不会向远处移动，或者渐行渐远，而是始终与摄像机保持相同的距离。为此，必须要对所有导入的贴图的一些关键属性进行调整。具体的实现步骤是，首先选中所有导入的贴图，然后在对象Inspector面板中，将“Texture Type”域的内容从“Texture”改变为“Sprite 2D and UI”，然后取消“Generate Mip Maps”选择框的选中状态，然后单击“Apply”按钮。当完成这些操作之后，Unity会标记这个资源可以应用到二维对象上。它允许

将透明的背景应用到合适的地方（例如PNG图片精灵上），这对图像渲染也有很重要的意义，如图5.4所示。

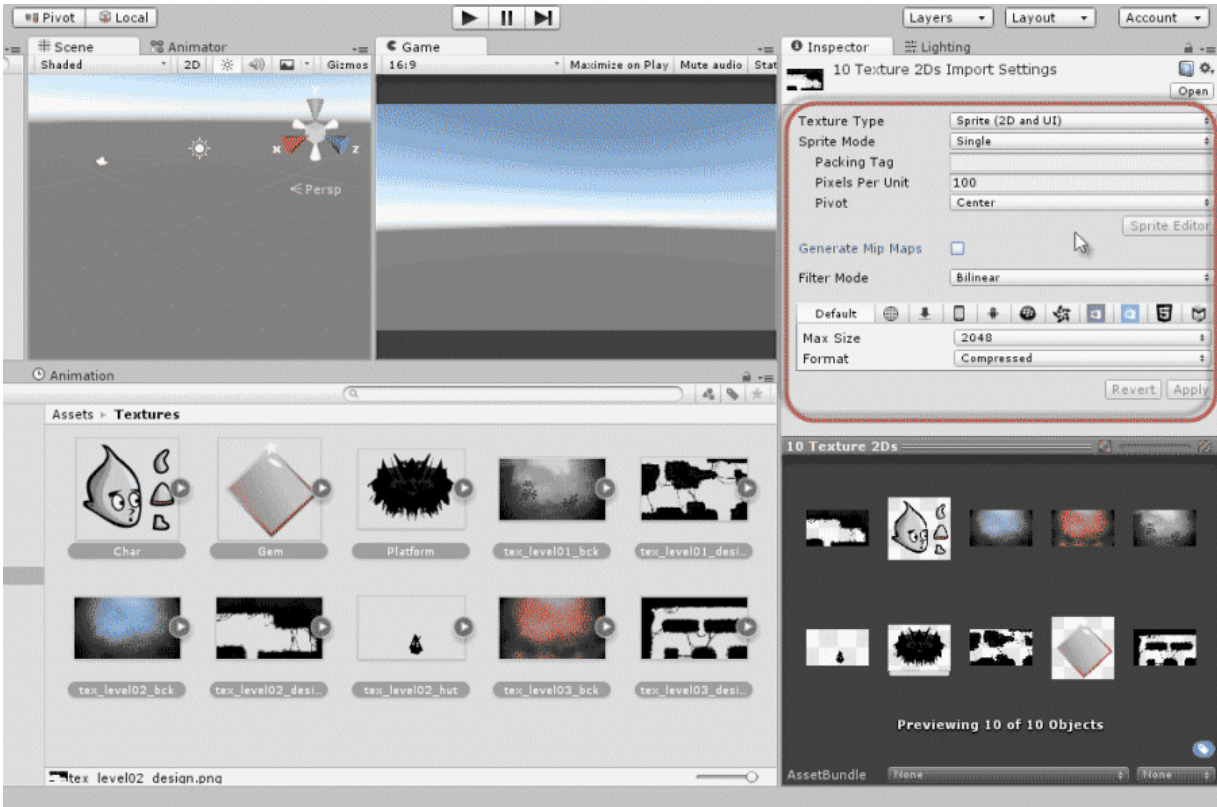


图5.4 对导入的贴图进行配置

现在已经导入了项目所需要的全部贴图。下面来配置主场景、游戏摄像机以及目标分辨率。首先切换到**Game**选项卡，然后将分辨率设置为1024×600，这个分辨率可以很好地应用在大量设备上。具体的步骤是，在Unity的工具栏上单击“Free Aspect”，然后在弹出的上下文菜单中选中“1024 x 600”这个选项，如果没有找到这个选项，可以单击列表底部的“+”按钮来添加一个新的分辨率，如图5.5所示。

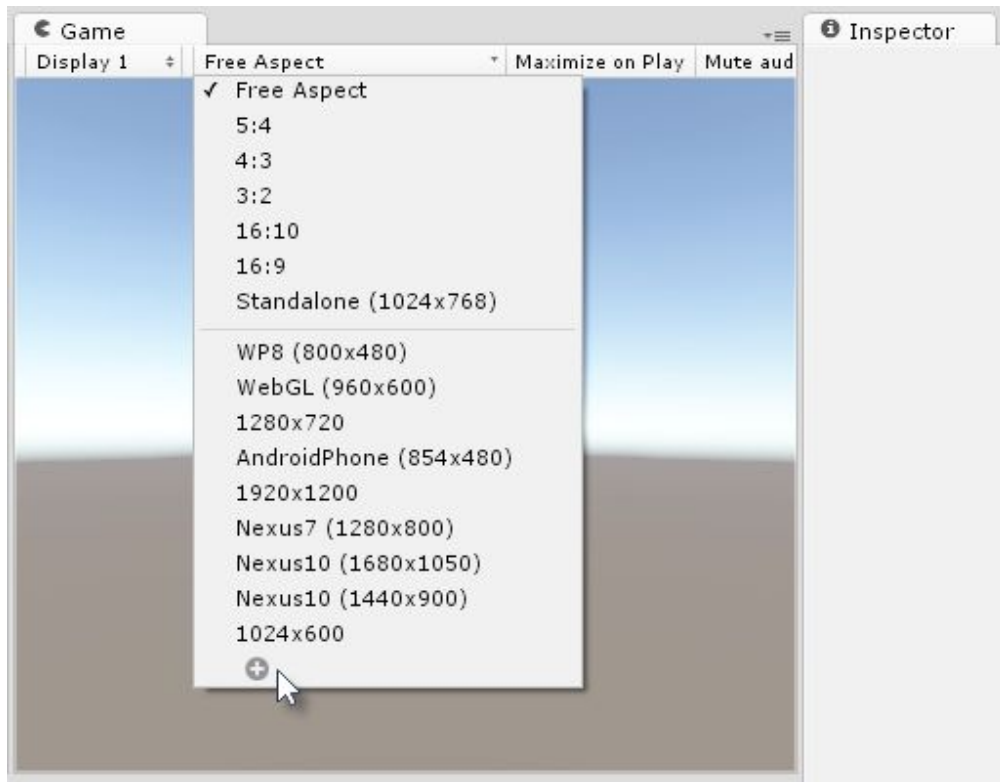


图5.5 添加一个新的游戏分辨率

在弹出的对话框中的名字处输入一个自定义的名字，然后在“Type”的下拉列表框中选中“Fixed Resolution”，最后在“Width & Height”后面的文本框中添加需要的分辨率的值。完成之后，单击“OK”，自定义的分辨率就会被添加到Unity中的可选项中，如图5.6所示。

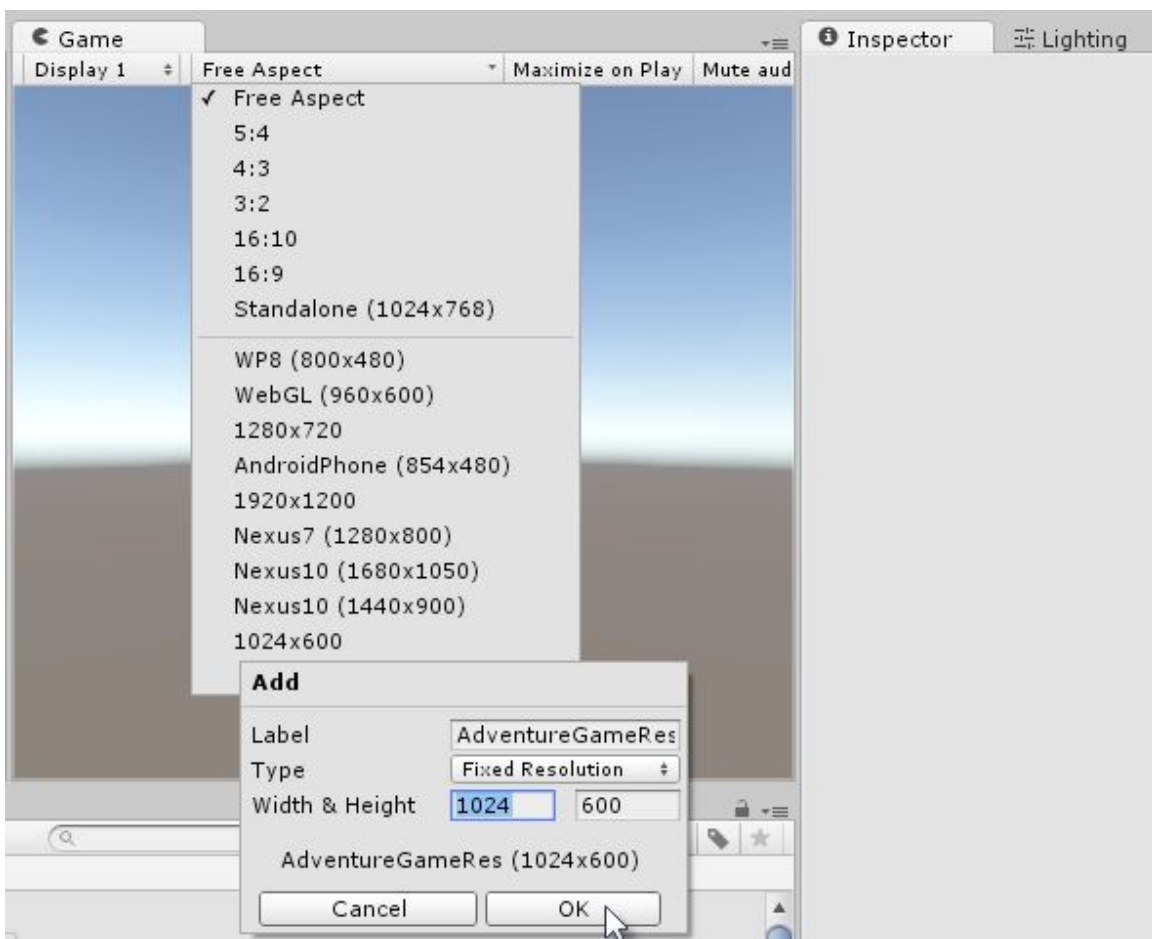


图5.6 创建自定义的分辨率

接下来，将场景中的摄像机配置为二维工作模式，这样当贴图以图像精灵的形式添加之后，就会以1:1的比例显示在屏幕上。首先，在场景选中主摄像机，可以在场景视图单击它或者层次（Hierarchy）面板中选择它。然后，在对象检查（Inspector）面板中将“Projection”的值改为“Orthographic”。这样可以确保对象以一个真实的二维效果显示，而没有任何的透视效果。最后将摄像机的Size的值设置为3。这个域值的公式为Screen Height / 2 / Pixel to World，在本例中，“Screen Height”的值为600， $600 / 2 = 300$ ， $300 / 100 = 3$ 。这里的100指的是应用在图像精灵贴图上的像素坐标与世界坐标的转换率，表示世界坐标中的一平米对应贴图

上的多少个像素，这个值为1表示1个像素对应着1米。可以通过在“项目（Project）面板”中选择一个图像精灵，然后在对象检查（Inspector）面板的“Pixel to World ratio”中修改这个值，如图5.7所示。

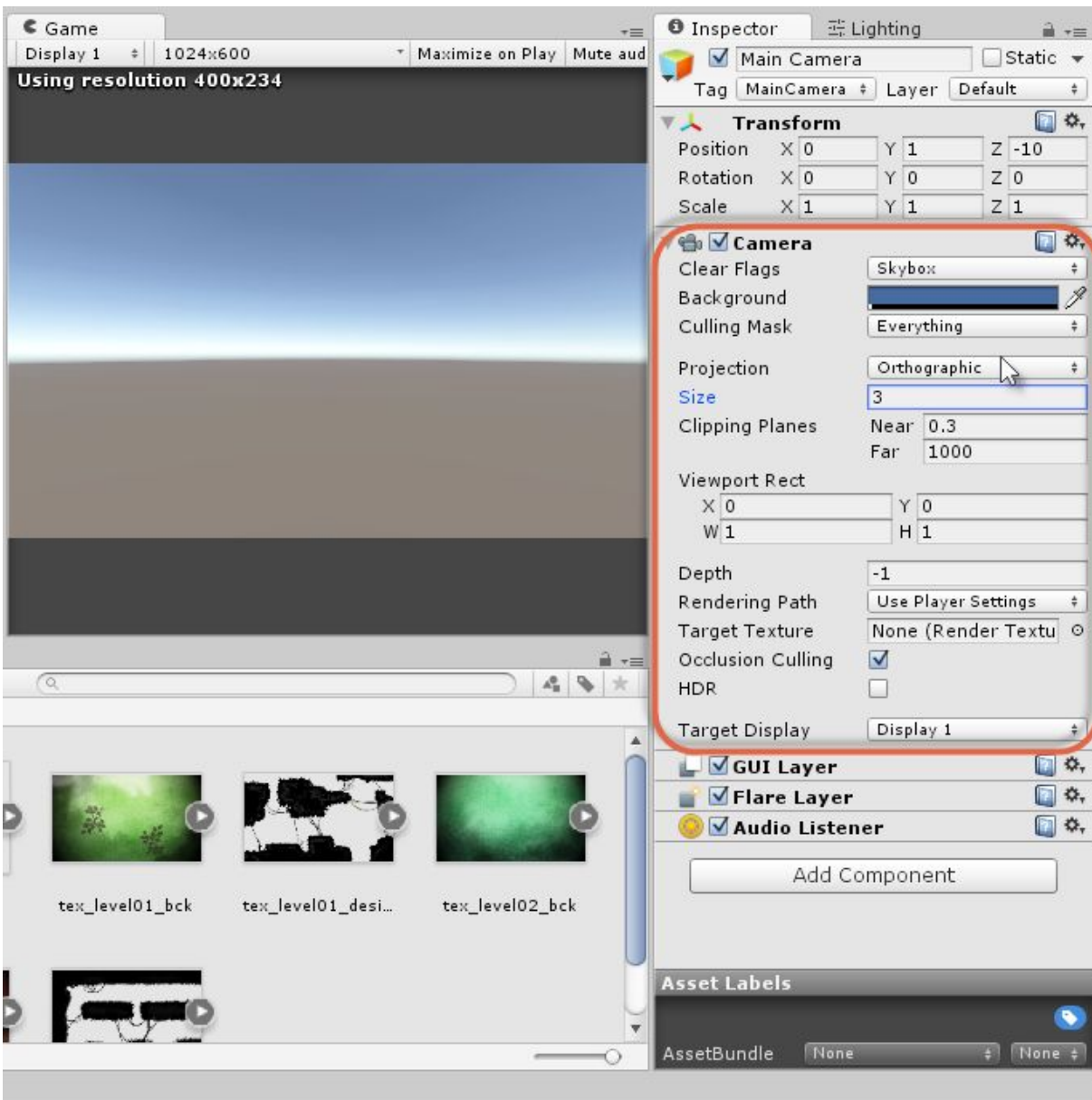


图5.7 配置Orthographic摄像机的“Size”



当对摄像机和场景中的设置进行测试时，只需要从项目（Project）面板中将一个背景贴图拖曳到场景中来，背景贴图的大小应该精确为1024×600，这样才适合场景的背景。因此，当背景被添加到场景，而且摄像机也被正确地配置之后，背景贴图就会填满整个屏幕，如图5.8所示。

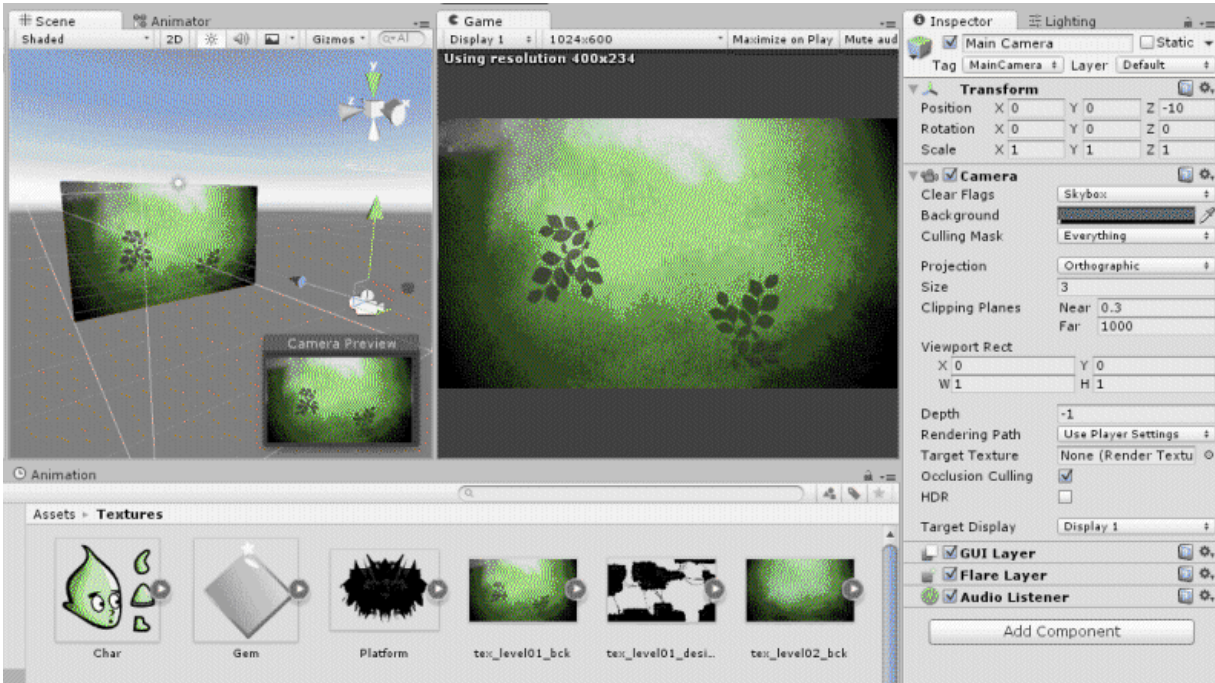


图5.8 使用贴图测试摄像机的设置

## 5.3 开始创建游戏的环境

本游戏要包含3个独立，但又相互关联的场景，玩家在场景中探险，然后从一个场景移动到下一个场景。玩家可以在不同的场景中进行切换，每当走到一个场景的边缘的时候，就可以进入到下一个场景。每个场景主要包括平台（Platforms）和台阶（Ledges），当然有时也会充满了危险与障碍。就图形资源而言，每个场景有两个贴图或者图片精灵

——前景和背景。图5.9和图5.10所示为场景1的实例，图5.9就是背景，而图5.10就是前景，其中包含了一个玩家必须穿越的完整平台和台阶。这些文件可以在本书配套文件的“Chapter05/Assets”文件夹中找到。



图5.9 场景的背景——tex\_level01\_bck.png

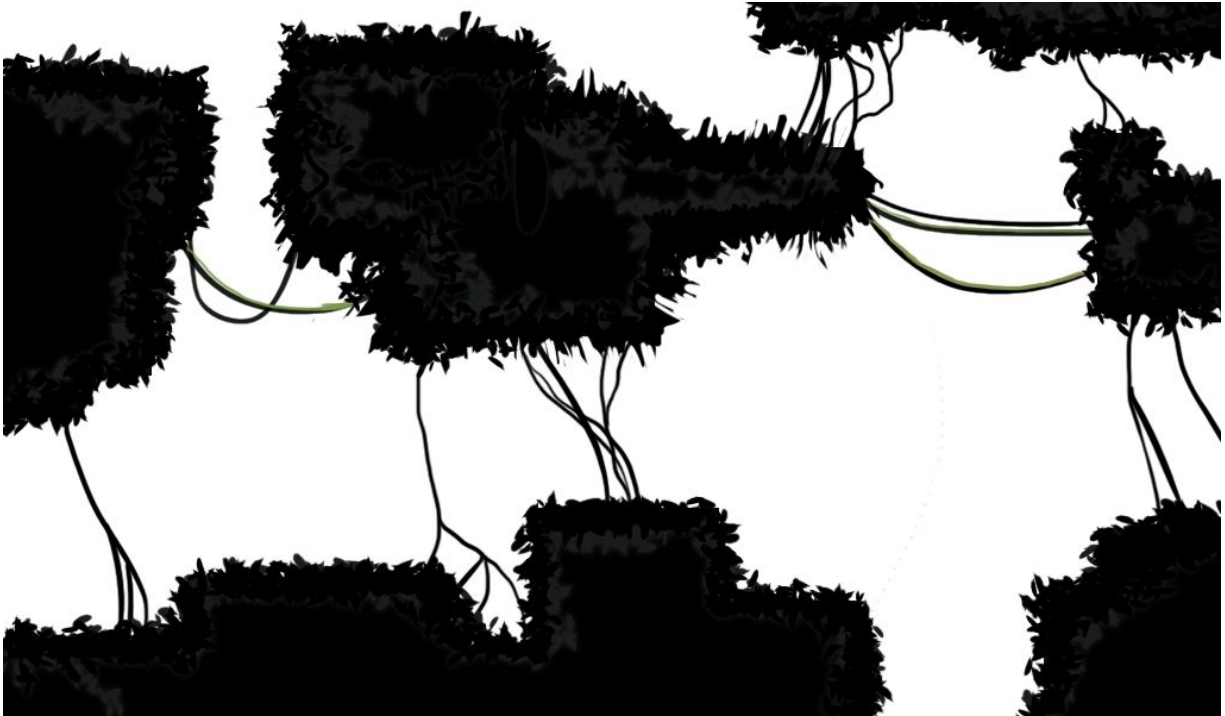


图5.10 Scene的前景——tex\_level01\_design.png

根据图5.9和图5.10创建第一个关卡，既可以使用现有的空场景，也可以创建一个新的场景，但是要确保场景摄像机按照其原有的大小来显示贴图。然后，将项目（**Project**）面板上的前景和背景图像精灵拖曳到场景中。这两者都会作为独立的图像精灵对象添加到场景中，最后将它们的位置都设置在（0,0,0），如图5.11所示。

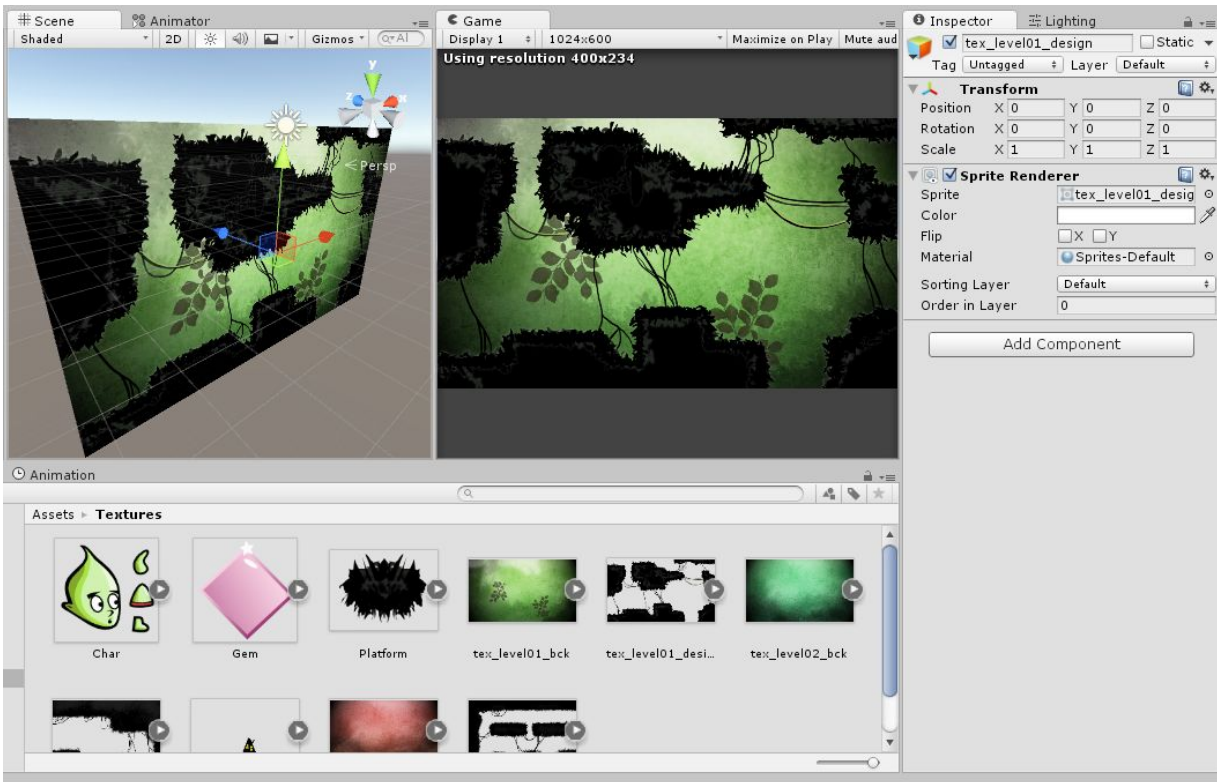


图5.11 向一个场景中添加一个背景和前景



如果将前景和背景贴图一起选中之后从项目（Project）面板拖曳到场景中，当松开鼠标时，Unity会询问是否想要创建一个动画。在这个案例中，Unity会假设要创建一个动画图像精灵，每一个选中的贴图都将成为动画序列中的一帧。如果不想这样做，则可以将每一个图像精灵拖曳到一个单独的层次（Hierarchy）面板，允许前景和背景可以同时看到。

现在两个图像精灵对象都被添加到了场景中，而且二者的世界坐标是相同的，都是（0,0,0）。问题是这两个图像精灵是相互重叠的，哪一个图像精灵应该位于前面呢？如果保持原样，现在的层次顺序就出现了冲突和混乱，Unity也不能始终自行作出正确的选择。有两种解决这个问题



题的方法：第一种是调整图像精灵Z轴上的坐标，使它与正交（Orthographic）摄像机的距离变近；另一种方法是在对象检查（Inspector）面板中改变Order的设置。Order值大的图像精灵意味着要在Order值小的图像精灵前面。这里将使用这两种方法，如图5.12所示。但是，注意Order优先性要高于位置。这就意味着Order值大的对象总是出现在Order值小的对象上面，即使将Order值大的对象的位置设置到了Order值小的对象的后面。

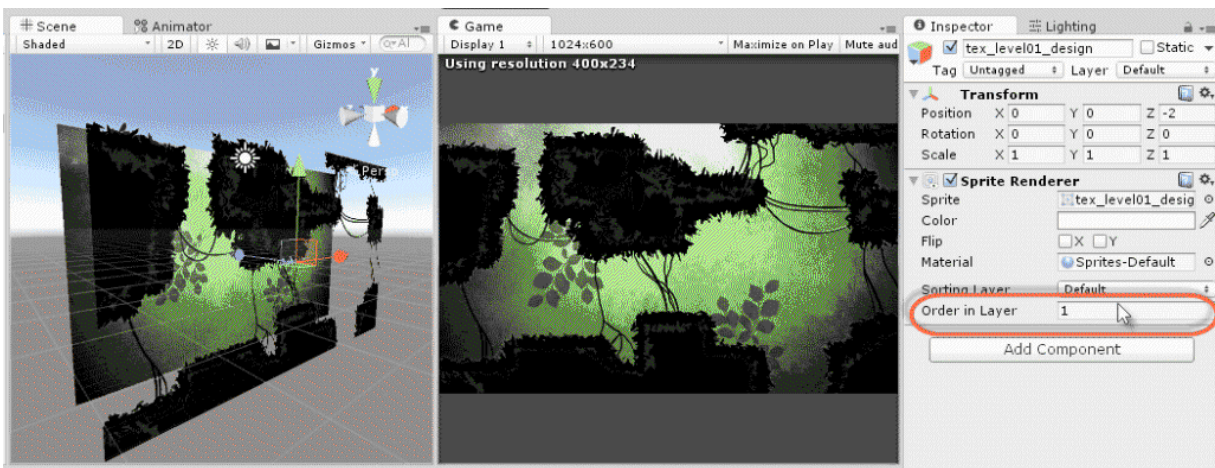


图5.12 在场景中调整图像精灵的顺序

在进一步进行游戏开发之前，要对场景中的层次结构进行组织，以避免后期出现过于复杂和混乱的情况。依次选中环境中的每一个对象，并给每一个对象赋予合适的名称，为背景起名为“scene\_background”，为前景起名为“scene\_foreground”。接下来，创建一个新的空对象，并将该对象命名为Env（Environment的缩写），它将是环境中所有静态对象的根父对象。这样就可以轻松地将所有相关对象归于一类。可以在应用程序菜单处依次选中“GameObject | Create Empty”，将创建好的对象放置在游戏世界的原点，然后将背景和前景对象进行拖曳操作，使它们成为这个对象的子对象，如图5.13所示。

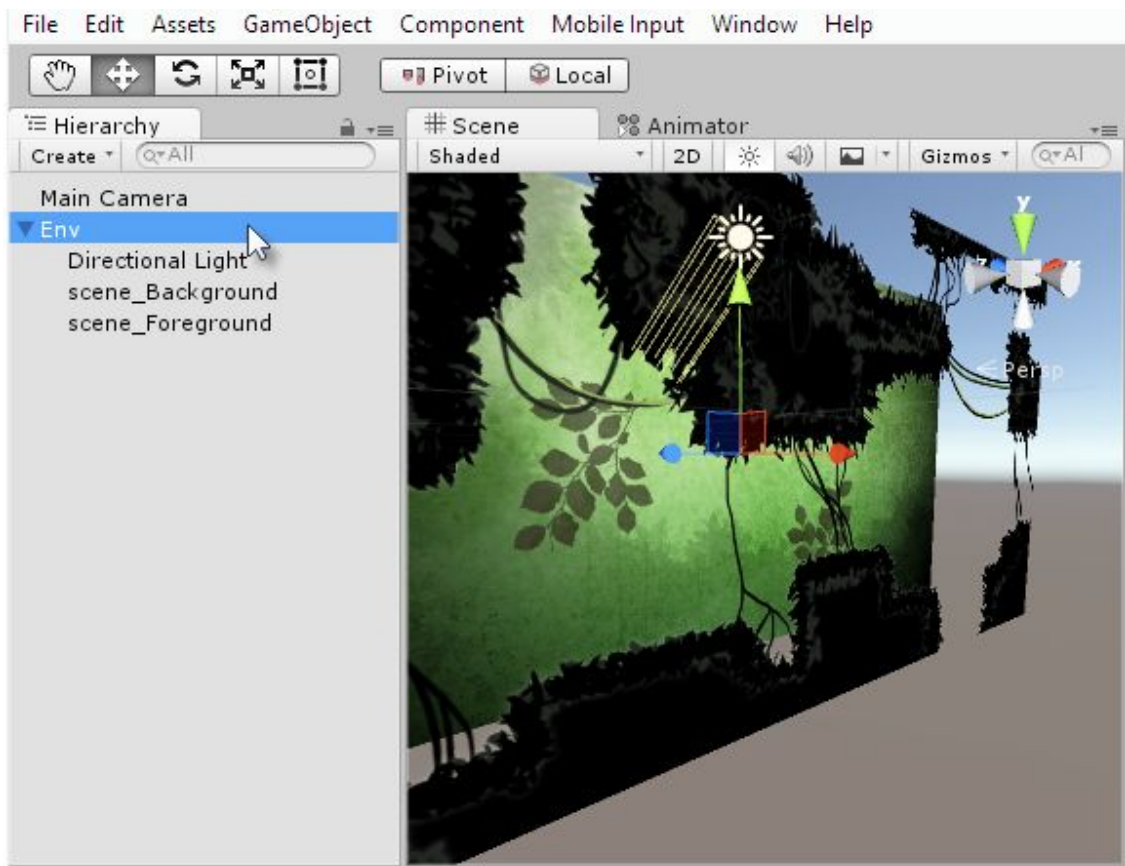


图5.13 对场景的层次进行组织

通过切换到游戏选项卡，可以得到一个关卡的初步预览，因为这些画面的感觉会影响到游戏者的情绪。这种感觉可以通过进一步添加摄像机的后期特效来加强。这些特效指的是可以被应用到相机上的基于像素的效果。它们会在每一帧对图像进行渲染，最终加重了游戏的气氛。在特效包中包含了图像的特效，这个特效包在项目创建的初期就已经导入过了。如果没有完成这个特效包的导入操作，可以通过依次选择菜单栏上的“Assets | Import Package | Effects”选项实现，“Image Effects”包保存的位置位于“Standard Assets/Effects”文件夹，导入之后，可以在应用菜单栏上依次选中“Component | Image Effects”来将选中的图像特效添加到选中的摄像机，如图5.14所示。



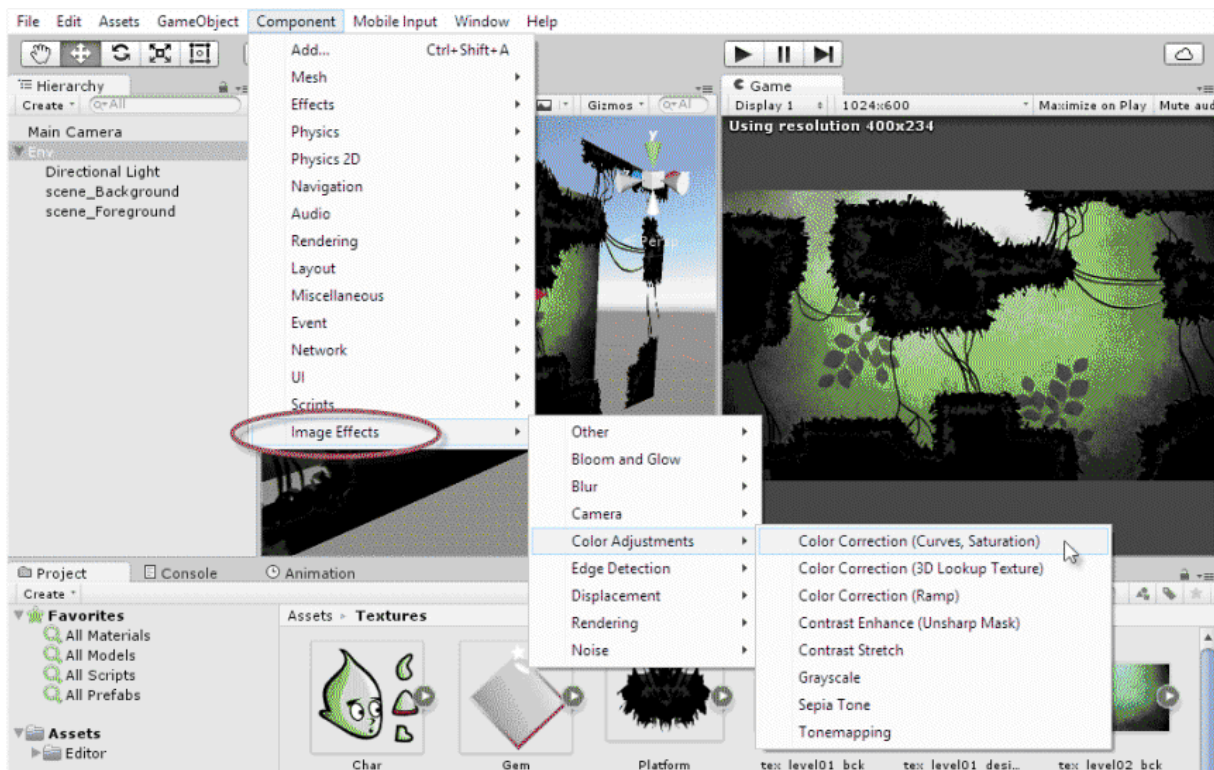


图5.14 向选中的摄像机添加图像特效

对于现在的这个关卡以及整个游戏中所有其他关卡，会向其中添加两个图像特效，即“Bloom Optimized”和“Noise and Grain”。添加之后，还需要在对象Inspector面板中对“Sliders”和“Settings”两个属性进行修改，这样才能够实现预期的效果。需要在游戏（Game）选项卡中不断对效果进行预览，因为在场景（Scene）选项卡中是看不到图像效果变化的。在许多情况下，这些设置将会多次地进行尝试，才能得到预期的效果，如图5.15所示。

到目前为止，场景中已经包含了从贴图文件中导入的前景和背景，而且也应用了图像特效资源包中的特效。这是一个不错的开始，但是还有很多工作需要去完成，让我们继续努力吧。

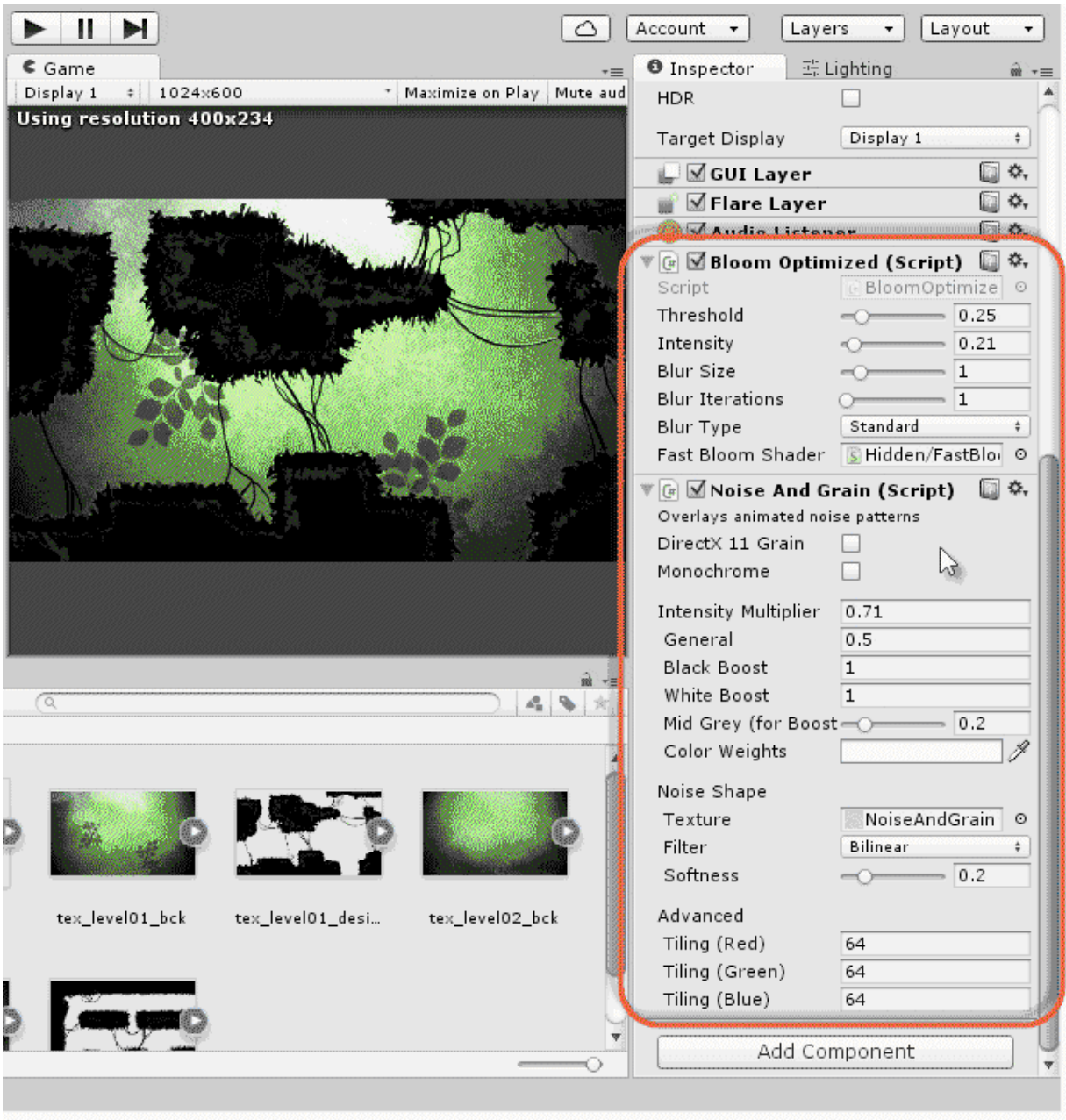


图5.15 应用在游戏摄像机上的图像特效

## 5.4 环境物理学

现在关卡最主要的问题是还缺乏互动性。确切地说，如果将一个玩家对象拖放到这个关卡时，然后按下工具栏上的“Play”键，这时会发现

玩家将从地板和墙壁中穿越而过，这是因为前景贴图并不会被Unity引擎看做是一个实体，它只是一个贴图，仅仅是一个表象，而并非是实质上的存在。这一节将会引入物理学和碰撞对此进行完善。首先创建一个临时的玩家对象（这不是最终版本，只是一个用来进行测试的临时白盒版本），在应用程序菜单上依次选中“GameObject | 3D Object | Capsule”，在场景中创建一个胶囊对象，将Transform组件中的Z值进行设置，使之与前景贴图相匹配（这里设置为-2），生成这个对象之后，将胶囊碰撞体组件从对象上去掉。默认情况下，胶囊对象上都会被分配一个三维的碰撞体（例如胶囊碰撞体），这个碰撞体主要用来实现三维世界中的物理属性，但是这是一个二维的游戏。如果想要移除这个碰撞体，在对象检查（Inspector）面板上单击“Capsule Collider”组件右上角的齿轮图标，然后在出现的菜单上选中“Remove Component”，如图5.16所示。

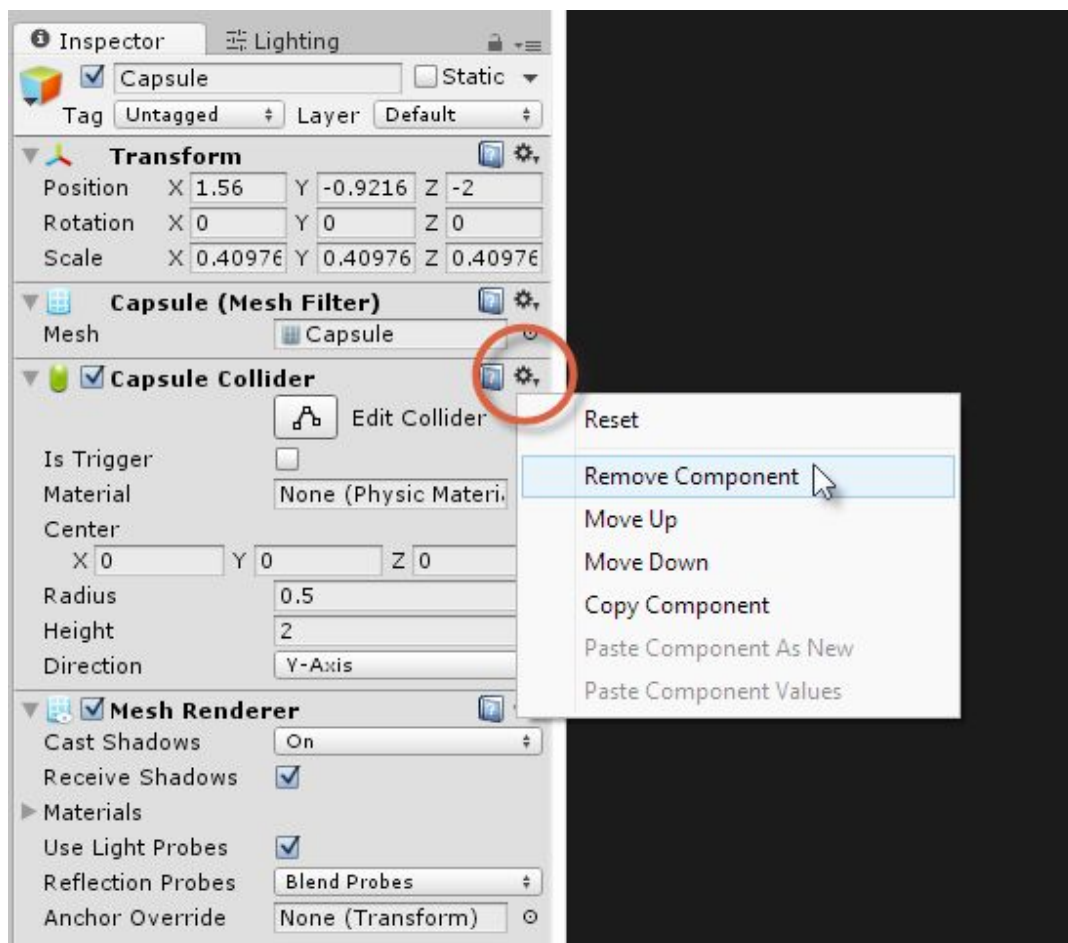


图5.16 将胶囊碰撞体组件移除

为了使这个对象具有二维世界的物理属性，需要从应用程序菜单处依次选中“Component | Physics 2D | Circle Collider”来添加一个圆形碰撞体组件，成功添加之后，在对象Inspector面板处修改圆形碰撞体组件的“Offset”和“Radius”值，这样就可以调整胶囊对象圆形碰撞体的大小和位置，使得它与玩家角色脚的大小相近似。如果希望更容易地对圆形碰撞体进行位置的调整，可以将场景（Scene）视图模式切换成“Wireframe”和“2D”模式。在视图工具栏使用“2D Toggle”按钮和“Scene Render”模式中的下拉按钮来实现模式的切换，如图5.17所示。



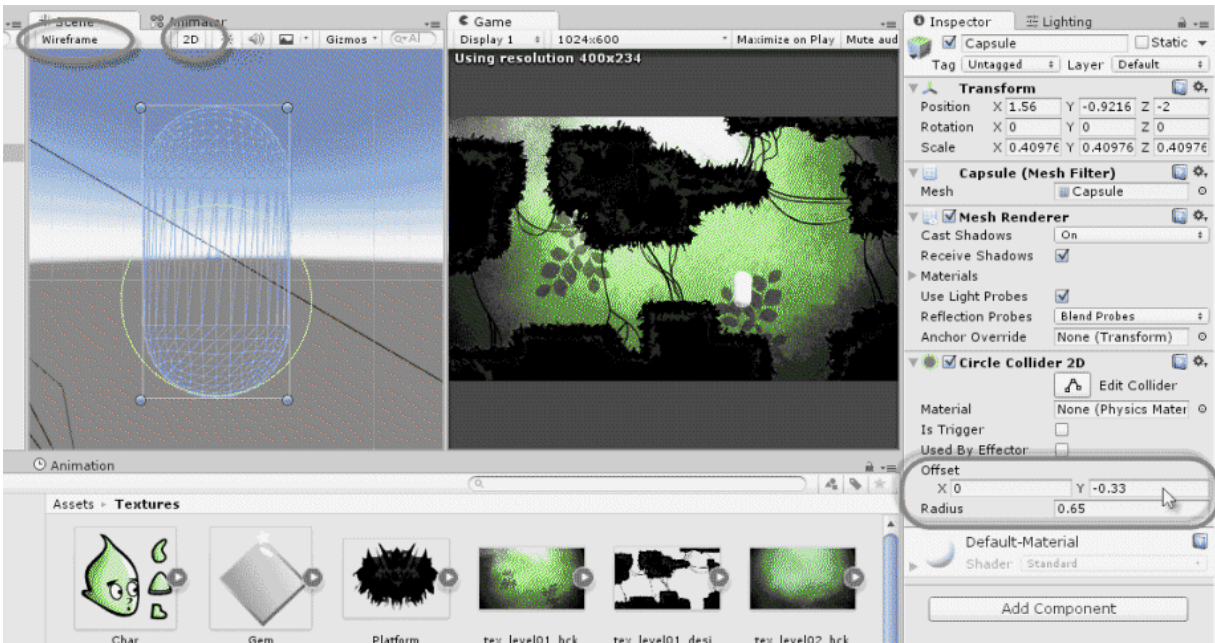


图5.17 对玩家角色的圆形碰撞体进行调整

接下来，如果需要对圆形碰撞体具有二维世界的物理属性，就得向胶囊对象添加一个刚体（Rigidbody）组件。首先，要在应用程序菜单中依次选中“Component | Physics 2D | Rigidbody 2D”，在游戏模式中对它能否正常工作进行预览。当单击“Play”图标时，胶囊对象就会在重力的作用下掉落，并穿过前景中的地面，如图5.18所示。

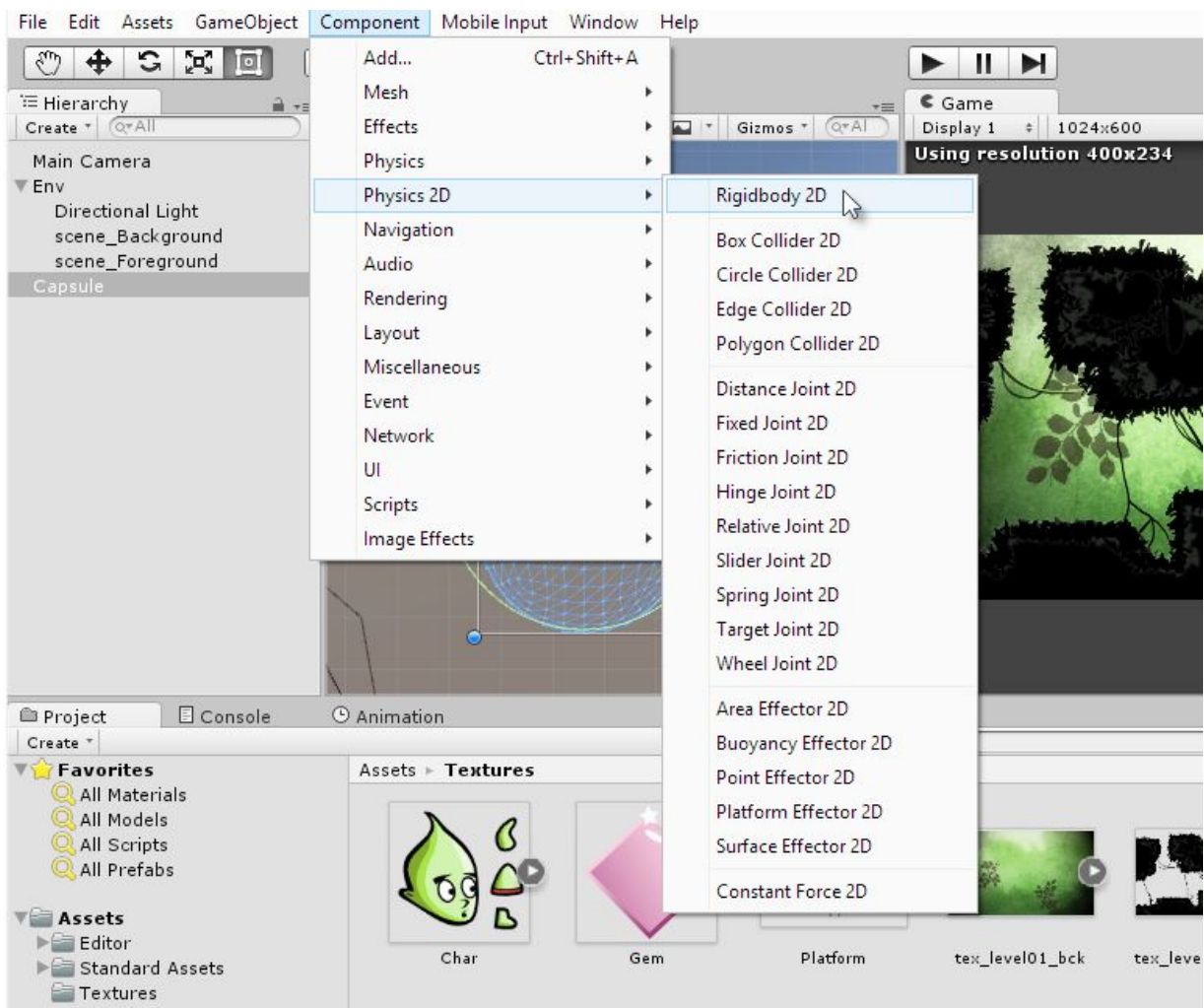


图5.18 向一个测试用的角色添加一个“Rigidbody 2D”组件

现在是时候将前景贴图作为一个统一的物理整体了。此时的测试玩家角色会穿过地面，当然这并不是所希望的。为了解决这个问题，需要为前景环境添加一个碰撞体。其中一种办法就是使用“Edge Collider 2D”，它可以帮助你围绕图像手动地画出一个与地形近似的多边形网格碰撞体。首先在场景中选中前景，然后从应用程序菜单中依次选中“Component | Physics 2D | Edge Collider 2D”，这样就可以为前景对象添加上一个“Edge Collider 2D”组件，如图5.19所示。



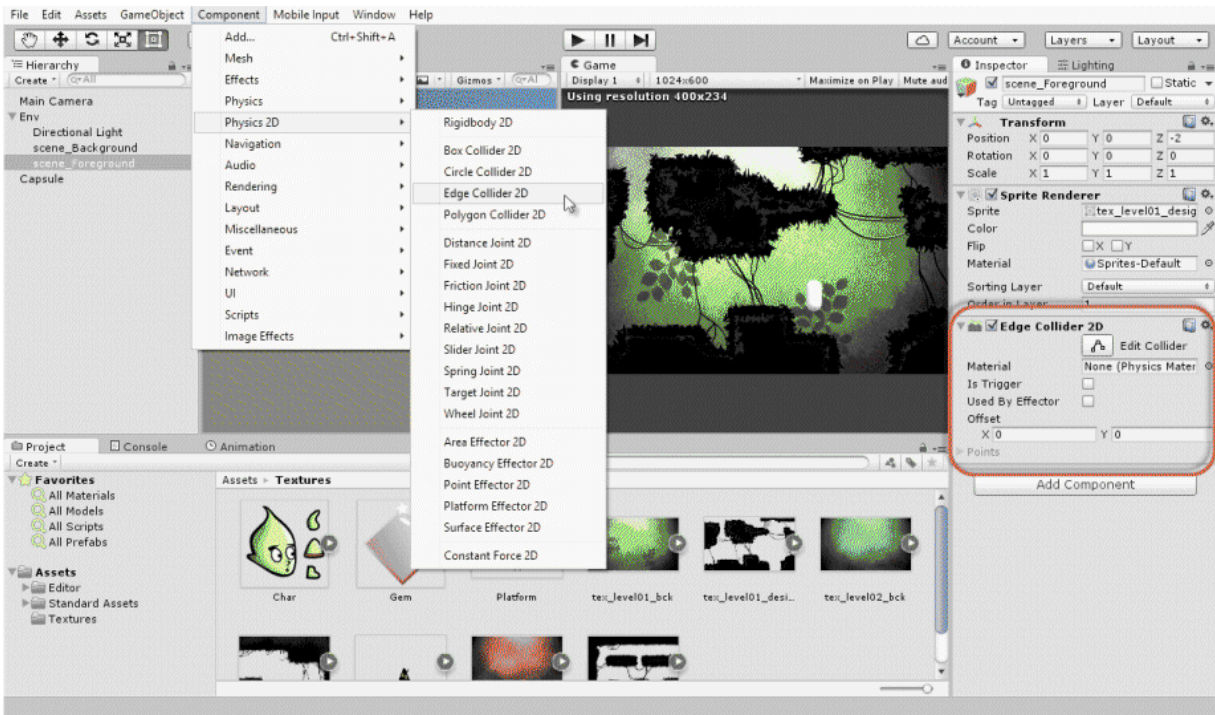


图5.19 添加一个“Edge Collider 2D”

默认情况下，一个添加进来的“Edge Collider 2D”对选中的对象和其他对象来说并没有什么明显的效果，仅仅是在场景中出现了一条单独贯穿整个屏幕的横线。当Gizmos工具按钮启用之后，在游戏（Game）选项卡中就可以看到这条线，或者在场景（Scene）选项卡中选中Foreground对象时，也可以看到这条线。如果玩家的位置位于这条横线的上方，在工具栏上按下“Play”键，玩家角色就会掉落到这条横线上，并将这条横线看作是一个固体的平面，如图5.20所示。

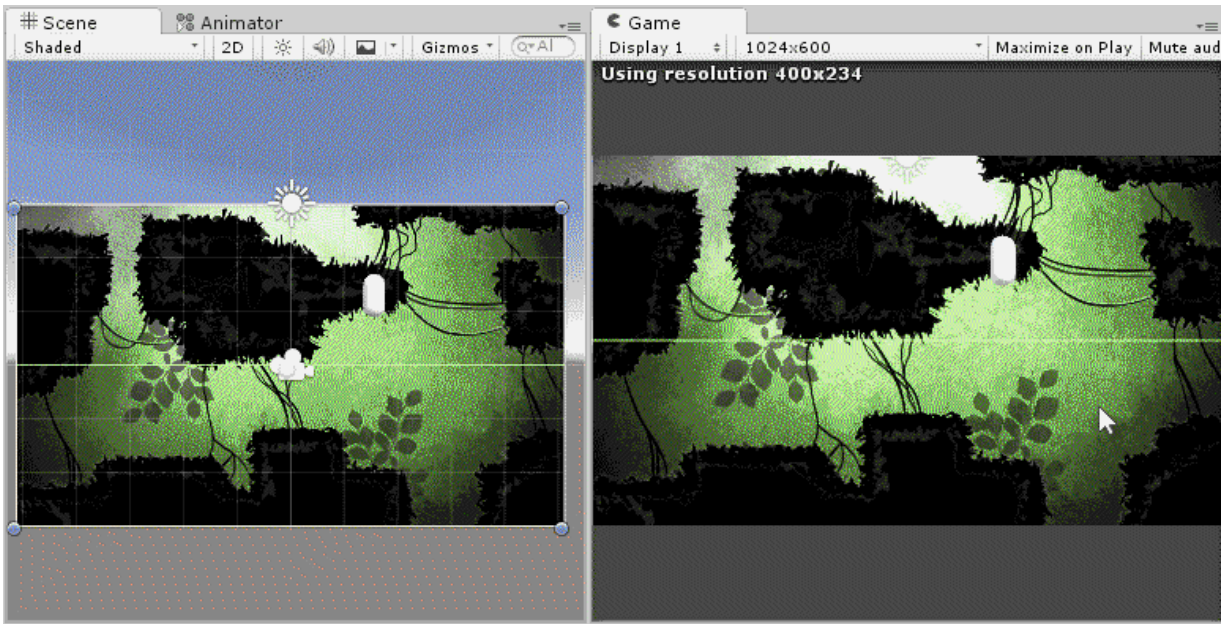


图5.20 “Edge Collider 2D”十分适合模拟地平面和固体表面

当然，地形并不是一个笔直的平面。这个地形既有起伏的地形，也有平坦的地形。使用“Collider Edit”模式可以将“Edge Collider 2D”组件绘制的与地形十分近似。可以在对象检查（Inspector）面板中单击“Edit Collider”按钮来进入这个模式，如图5.21所示。

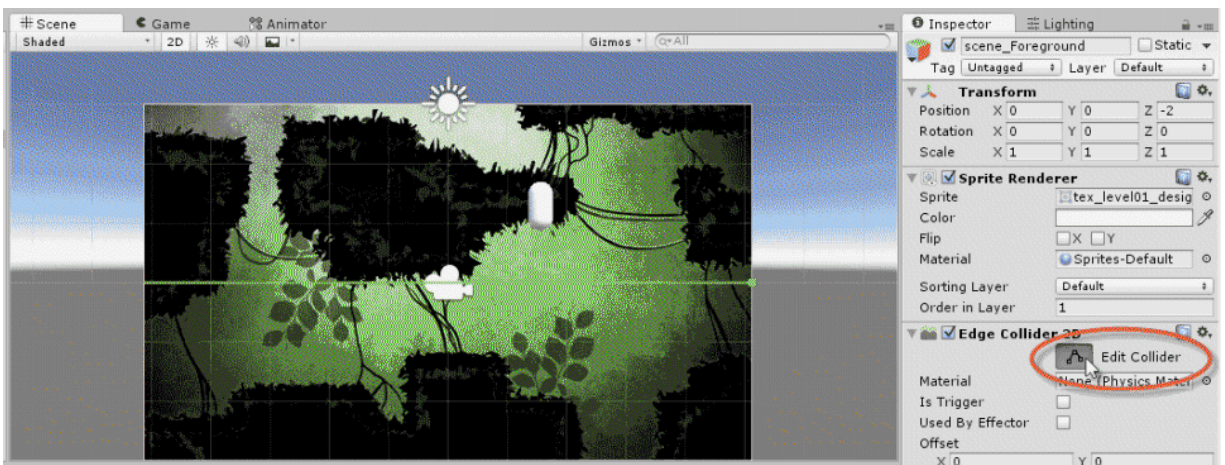


图5.21 使用“Edit Collider”模式修改“Edge Collider 2D”的形状

当“Edit Collider”模式激活之后，就可以改变碰撞体的形状使它适应地形。现在把注意力放在其中一块区域上，例如地形的右下角。把鼠标的光标移动到“Edge Collider”（绿线）边缘点的上方，然后单击这个点，就可以拖动它改变位置，将这个点拖动到场景的右侧，如图5.22所示。

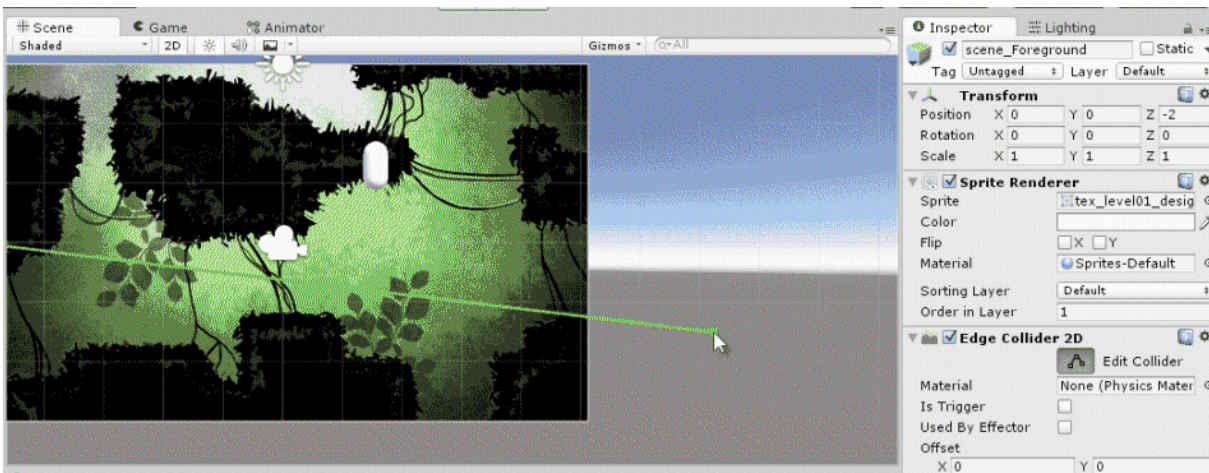


图5.22 对“Edit Collider”进行变形以适应地形

接下来，拖曳“Edit Collider”左侧的点来匹配右侧小岛最左边的点，如图5.23所示。



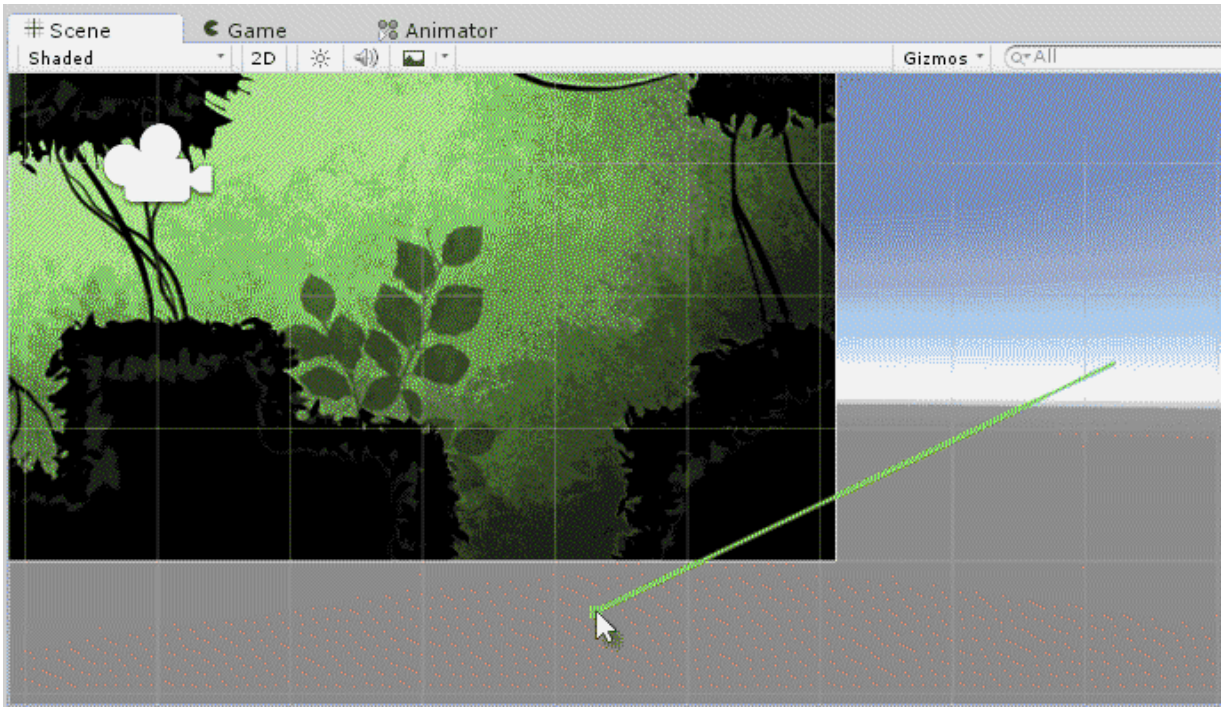


图5.23 定位到右侧小岛的左边缘点

现在，左右边缘点的位置已经确定了，再添加一些额外的点来改变它的形状，使其与右边的岛屿形状相吻合。将光标移动到直线上的任意一点，执行单击操作并拖动以插入一个新的点，将新点的位置与岛屿相匹配。不断重复这个过程，直到添加了所有勾画岛屿大致曲线所需要的点，如图5.24所示。

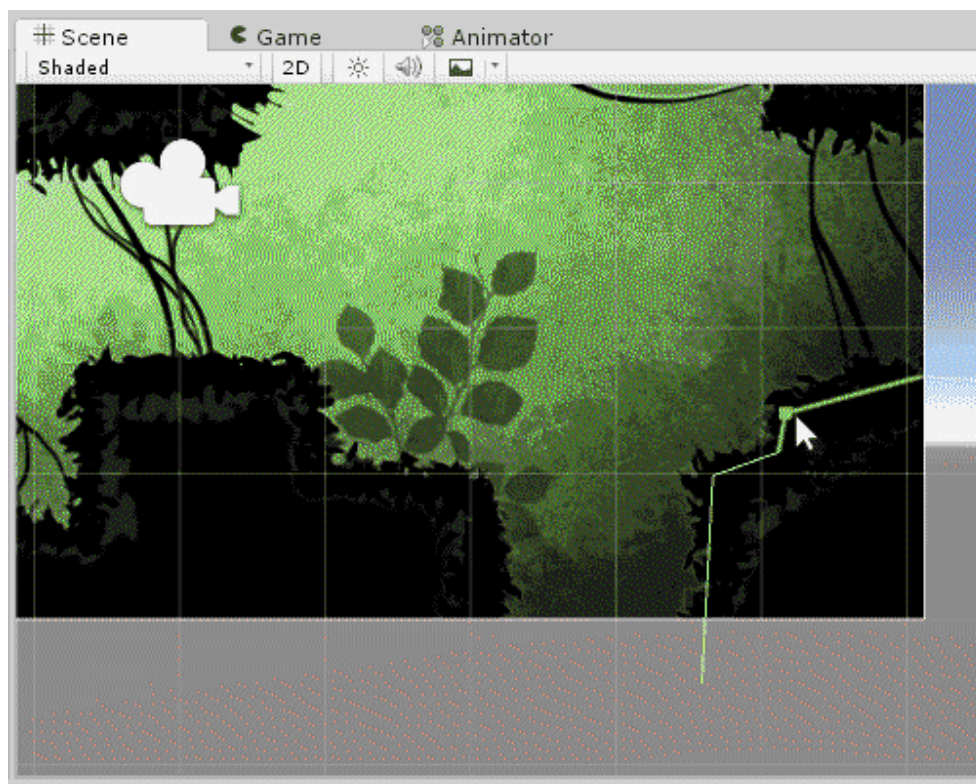


图5.24 打造一个与最右侧岛屿相匹配的“Edge Collider”

现在已经拥有了一个与最右侧岛屿地形相匹配的线条。创建完这个线条之后，在对象检查（Inspector）面板中再次单击“Edit Collider”按钮，这样就离开了“Edit Collider”模式。接着为图中的剩余岛屿创造碰撞体（Collider），向这个对象继续添加新的“Edge Collider”。可以向同一个对象添加任意数量的“Edge Collider”，每一个碰撞器都应该匹配完整地形结构中的一个独立岛屿，如图5.25所示。

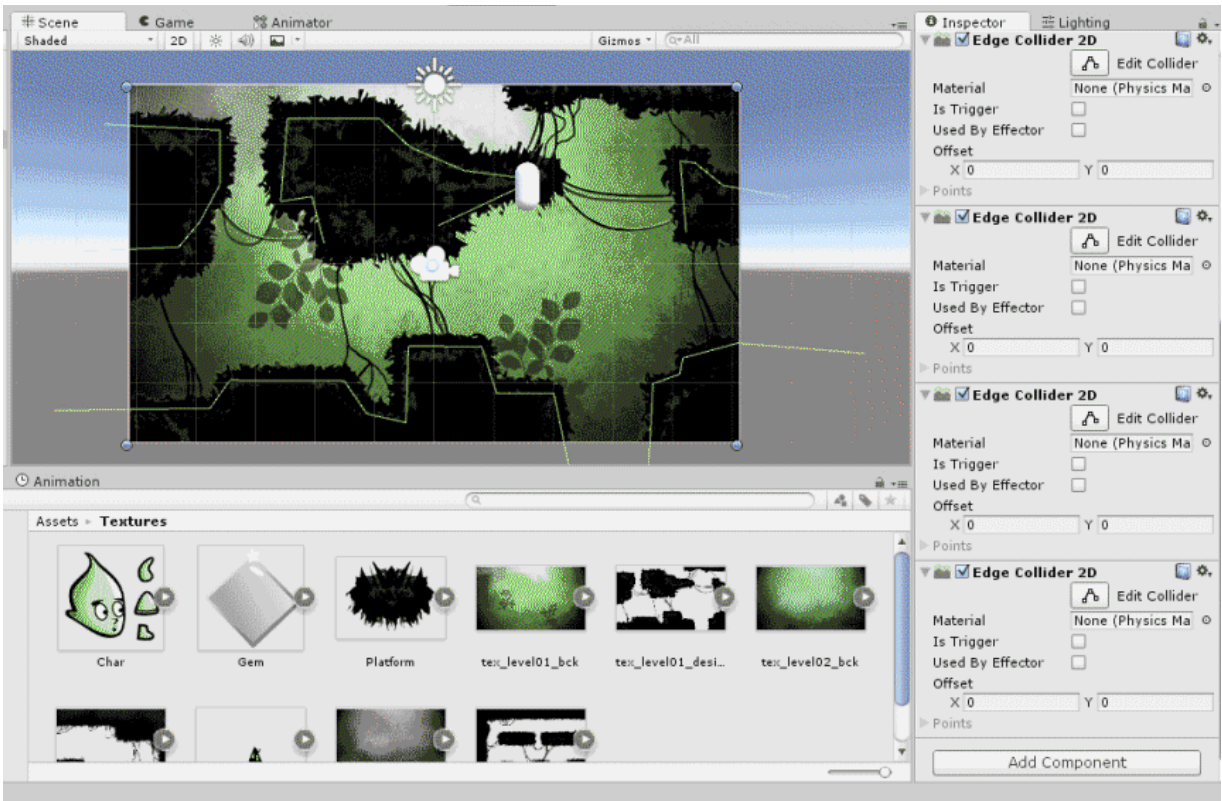


图5.25 在一个对象上应用多个“Edge Collider”来模拟复杂地形

现在已经将多个“Edge Collider”添加到了一个前景对象上，使它与场景中的地形相匹配。可以来测试一下，首先单击工具栏上的“Play”图标，这次可以看到玩家的胶囊对象与地形的交互。这一次，胶囊对象不会再穿过地面了，而是落在了地面上。这说明地形已经成为游戏物理系统中的一部分了，如图5.26所示。



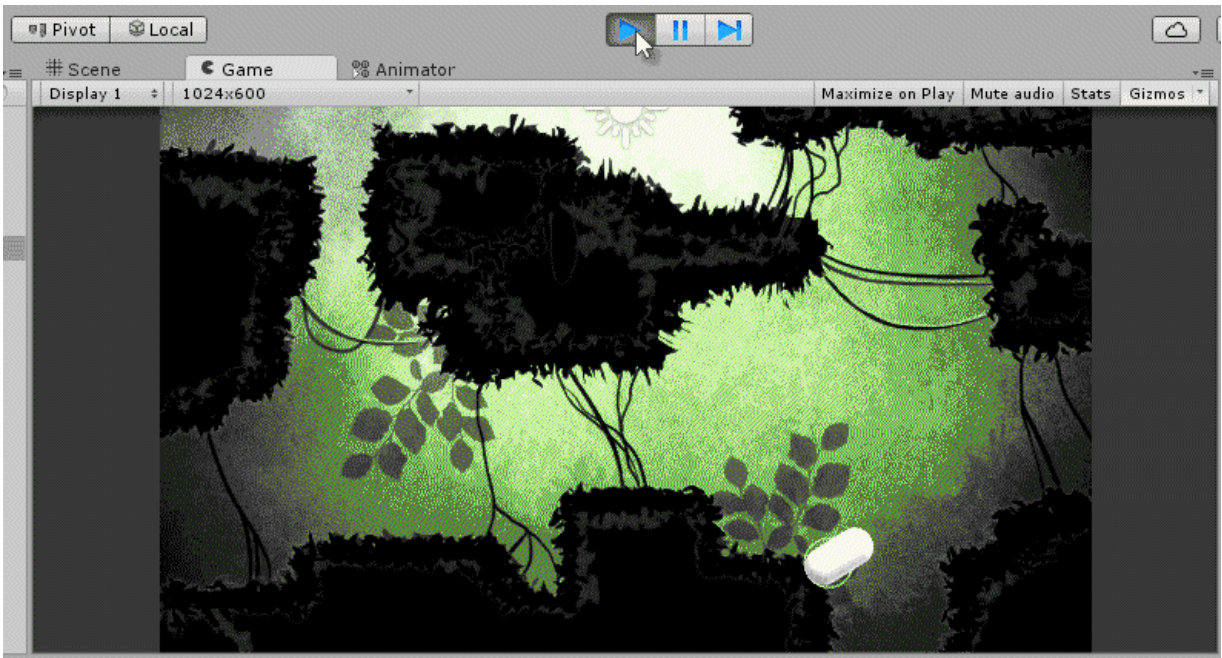


图5.26 胶囊对象通过“Edge Colliders”与地形发生交互

现在已经拥有一个使用了“Edge Collider”组件的完整地形，这个地形不仅仅是看起来跟预期一样，而且也已经成为玩家角色和其他物理实体的障碍。当然，到现在为止，一直在使用一个近似玩家的粗糙模型，现在是时候对其进行改进了。

## 5.5 创建一个玩家

游戏中的玩家（Player）角色是一个可以被控制的小型绿色生物，玩家可以通过使用大量的游戏设备来控制这个角色的行为，例如行走、跳跃、互动等。上一节创建了一个白盒（原型）角色来测试与环境之间的互动，现在要开发一个更加深入的玩家角色，图5.27所示为本章早些时候导入的角色贴图，这个贴图包含了玩家的肢体等部位。



图5.27 在同一张贴图中的玩家角色和它的四肢

图5.27所示的玩家贴图被称为贴图册（Atlas Texture）或者图像精灵表（Sprite Sheet），这是因为它在一张单个的贴图中包含了全部的帧或角色的所有部分。这个贴图的问题就在于，当将它从项目（Project）面板拖动到场景中时，会发现其变成了一个独立的图像精灵。这是因为Unity会将所有独立的部分看作是一个图像精灵。但是，这些部分应该被分成独立的单元，如图5.28所示。

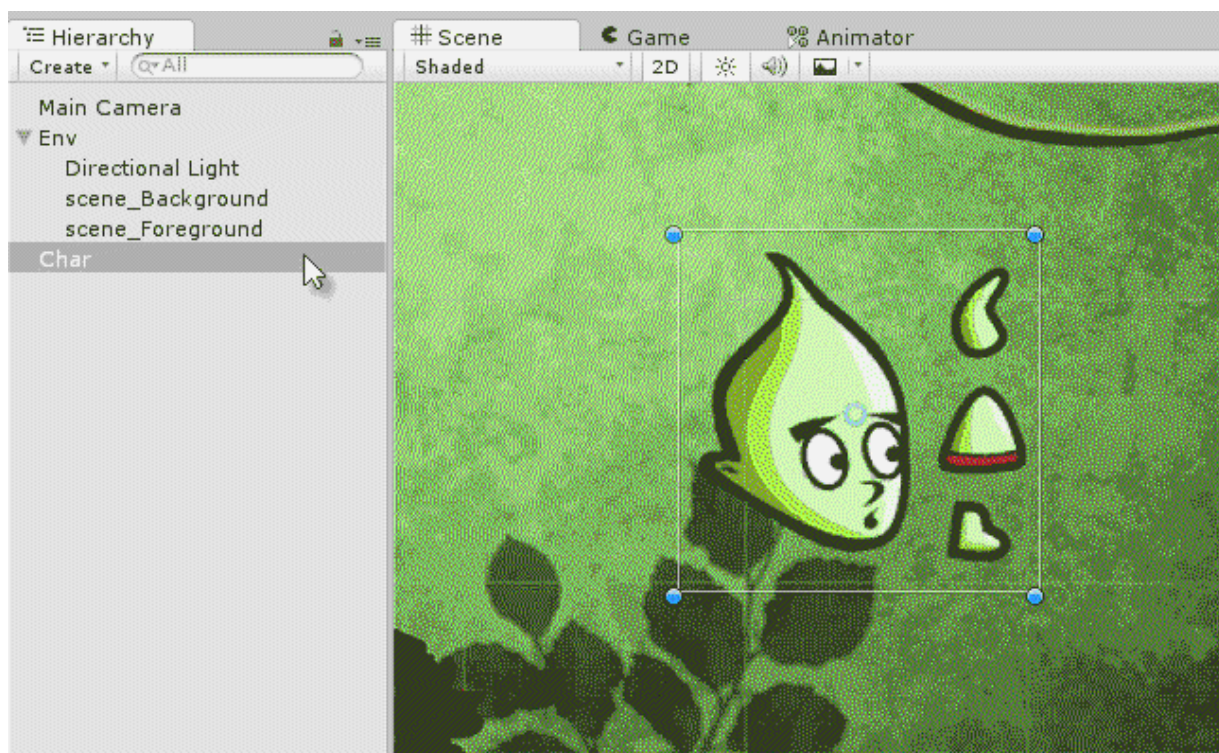


图5.28 玩家图像精灵贴图需要被分割成独立的部分



使用“Sprite Editor”将这个角色贴图按照身体的各个部位分割成独立的部分。使用这个工具时，需要首先在项目（Project）面板上选中角色贴图，然后在对象 Inspector 面板，将“Sprite Mode”的值由“Single”修改为“Multiple”，之后单击“Apply”键，最后单击“Sprite Editor”按钮来打开 Sprite 编辑工具，利用这个工具就可以将整个贴图分割成多个部分，如图5.29所示。

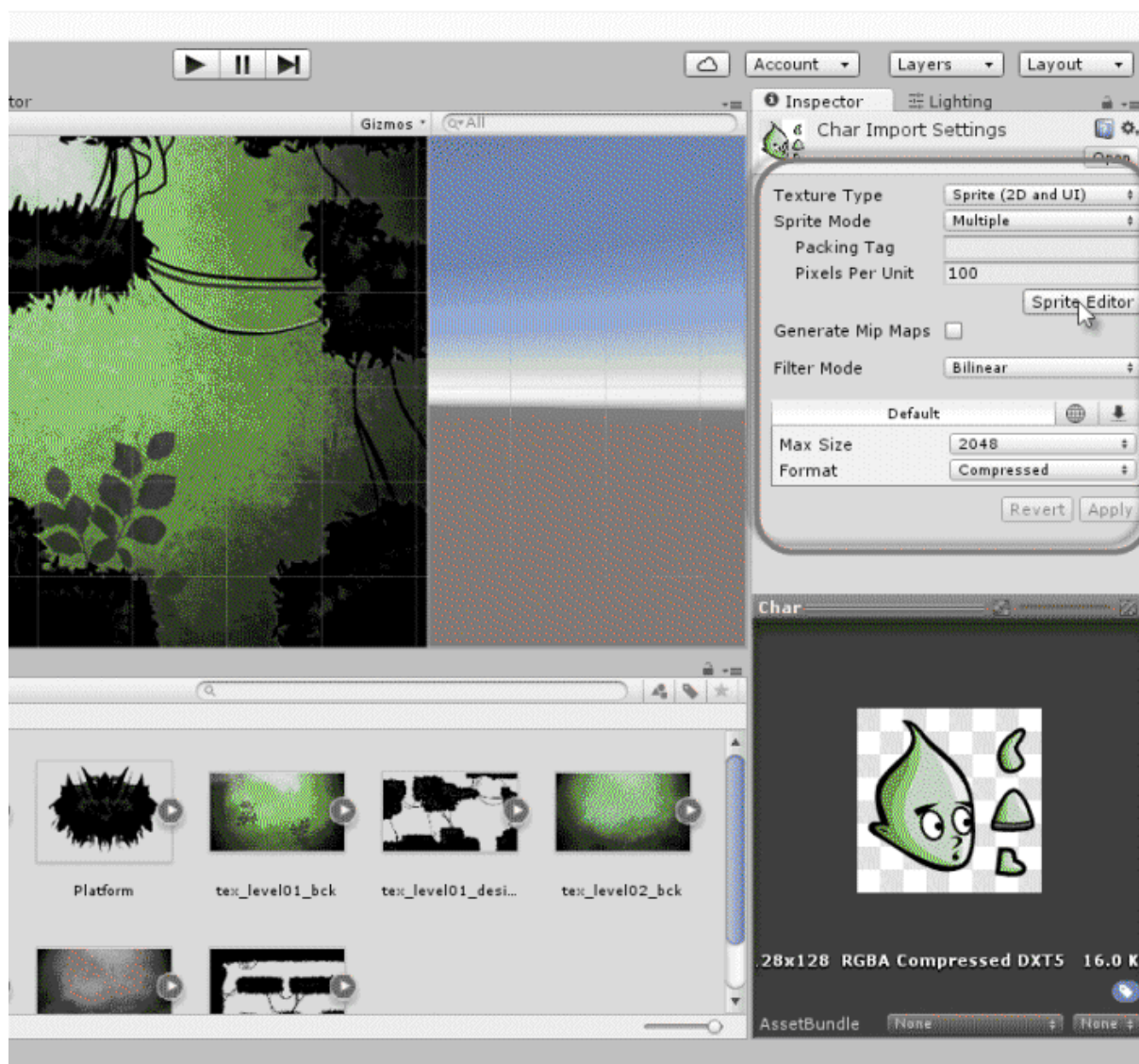


图5.29 将图像精灵的模式指定为“Multiple”

使用“Sprite Editor”工具就可以将一个贴图的各个部分分割成独立的、分离的单位。可以采用在贴图中的每一个应该独立的图像上画一个矩形，这个矩形可以通过使用鼠标在图片上拖曳出一个区域来实现，如图5.30所示。

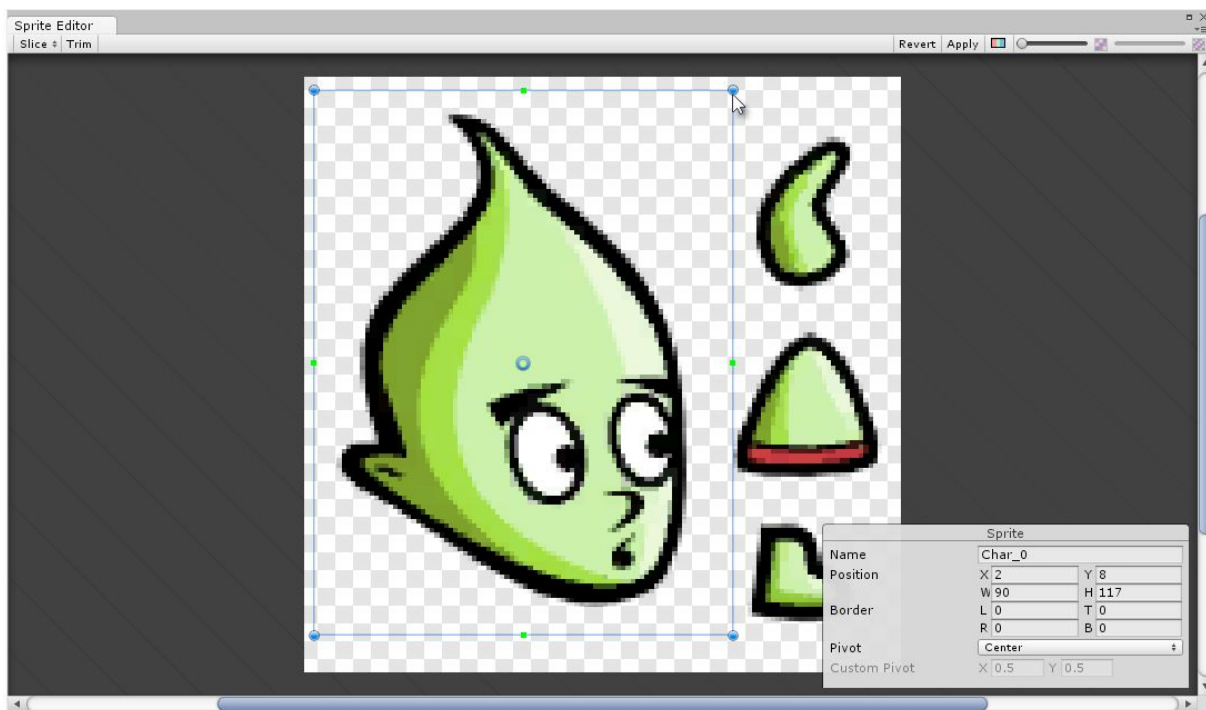


图5.30 手动分割图像精灵

现在已经实现利用手工的方法对贴图进行切割，不过Unity引擎还可以利用对孤立像素区域识别来自动地对贴图进行切割。也可使用这种方法：在“Sprite Editor”窗口的左上角单击“Slice”按钮，如图5.31所示。

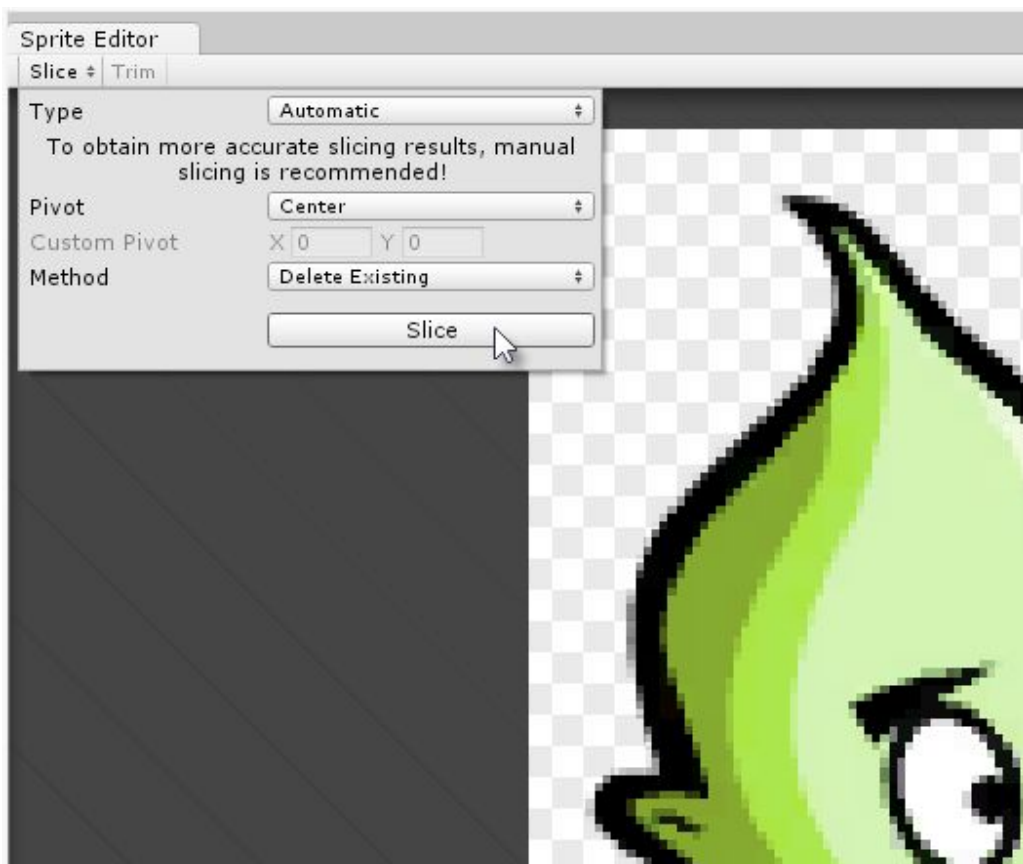


图5.31 使用切片（Slice）工具

在切片（Slice）工具窗口中，要确保“Type”被设置为“Automatic”，这意味着Unity引擎将会自动检测每个图像精灵的位置。“Pivot”的值应该保留为默认的“Center”，确定每一个图像精灵的轴心点。Method应该为“Delete Existing”，这意味着任何现有贴图的图像精灵或者Slice都将会被删除，并由新生成的切片（Slice）完全替换。然后，单击“Slice”按钮确认操作，贴图会被切成多个独立的具有清晰边界的图像精灵，如图5.32所示。

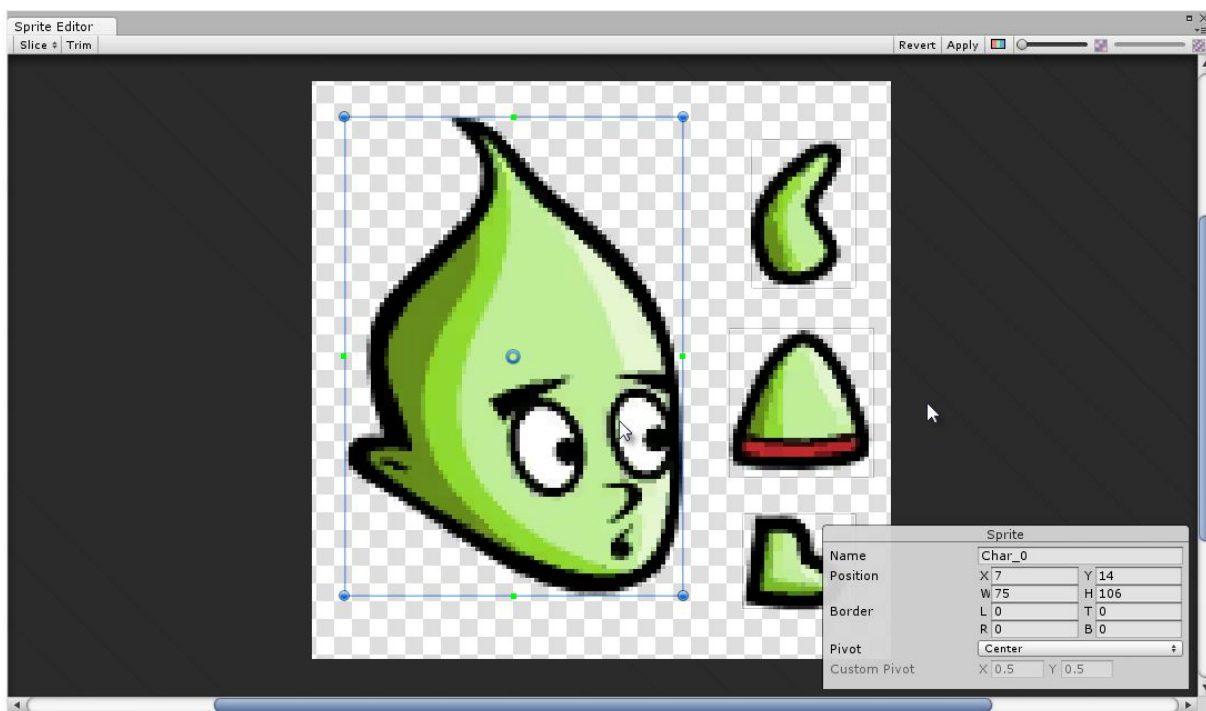


图5.32 切好的图像精灵

现在的贴图已经被分割成了多个图像精灵：头、躯干、手臂和腿。最终在场景中出现的玩家角色显然应该有两只胳膊和两条腿，它们需要通过复制图像精灵来实现。最后的过程是为图像精灵设置一个轴心点，图像精灵会围绕这个点旋转。这一点将来为角色设置动画是十分重要的。首先从为头部设置轴心点开始，在编辑器中选中头部图像精灵，然后拖曳枢轴手柄（蓝色圆圈）来重新定位图像精灵的旋转中心，将手柄拖曳到头的正下方，大概就是头与脖子相连接的地方。这么做是有原因的，因为头要以这一点为中心摆动。当移动支点的时候，应该可以在“Sprite Properties”对话框中的Custom Pivot区域看到X值和Y值的变化，这个对话框位于“Sprite Editor”右下角，如图5.33所示。



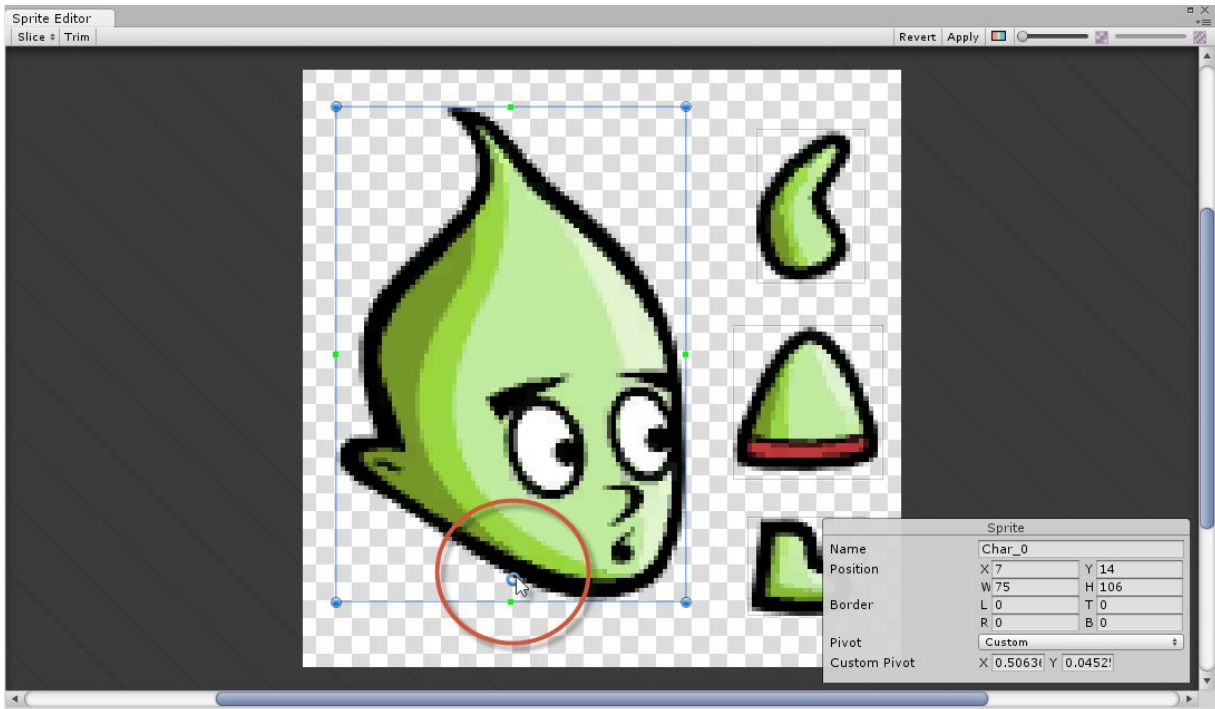


图5.33 重新定位图像精灵的轴心点

接下来，我们要为手臂定位轴心点，这个点应该位于肩关节也就是手臂与躯干连接的位置；对于腿来说，这个点应该靠近臀部，也就是腿连接到躯干的地方。最后，对于躯干来说，它的轴心点应该在髋关节位置，如图5.34所示。

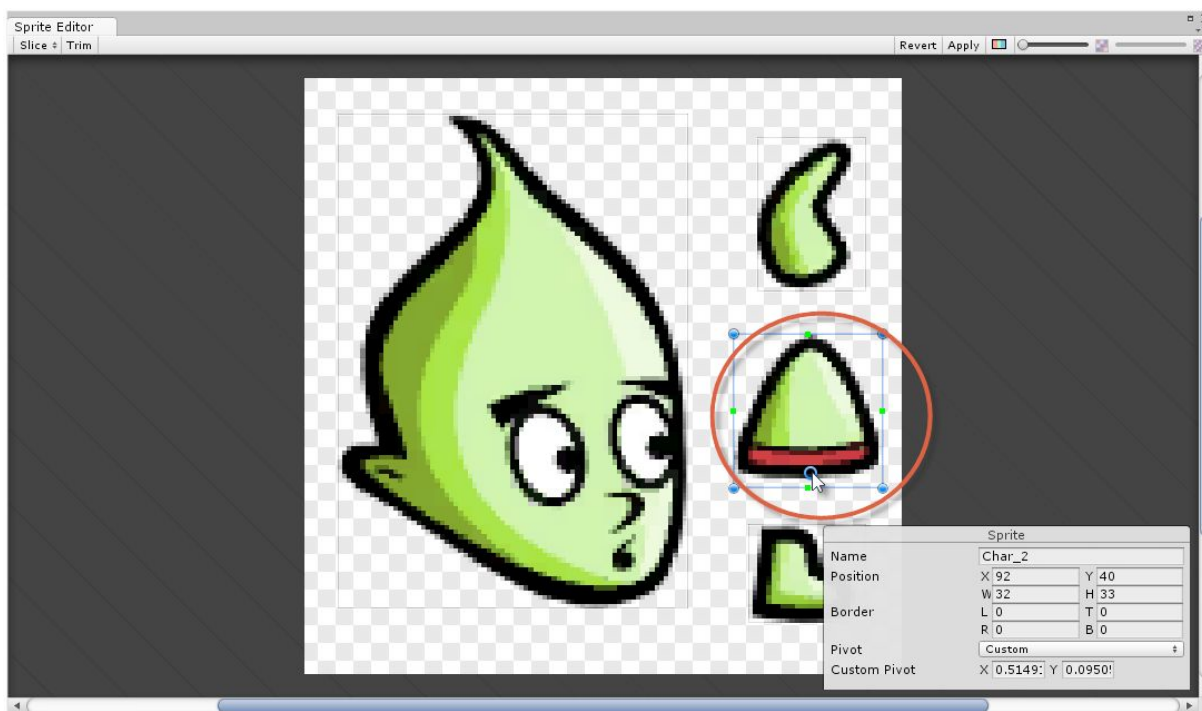


图5.34 为玩家角色的躯干定位轴心点

以上操作完成之后，我们单击“Apply”按钮来确认做出的修改并关闭“Sprite Editor”。现在我们回到Unity的主界面，就可以在项目（Project）面板中看到角色贴图的变化。具体来说，就是在角色贴图上手边部分多了一个箭头图标。当单击这个箭头图标的时候，这个贴图会向右展开，将包含的所有独立的图像精灵展开成一行，可以将这些图像精灵各自拖曳到场景中，如图5.35所示。



图5.35 对角色图像精灵进行预览

现在我们已经将贴图中所有玩家的图像精灵分离出来了，是时候来建立一个游戏场景中的角色了。首先在应用程序菜单处选中“GameObject | Create Empty”来创建一个空的游戏对象，并将这个对象命名为“Player”，在对象Inspector面板中为其Tag分配“Player”。这个对象将成为游戏中玩家角色最高的父对象。这个对象的子对象就是玩家角色的组成部分——躯干、手臂和腿。我们将躯干图像精灵从项目（Project）面板拖曳到层次（Hierarchy）面板使之成为玩家对象的子对象，如图5.36所示。

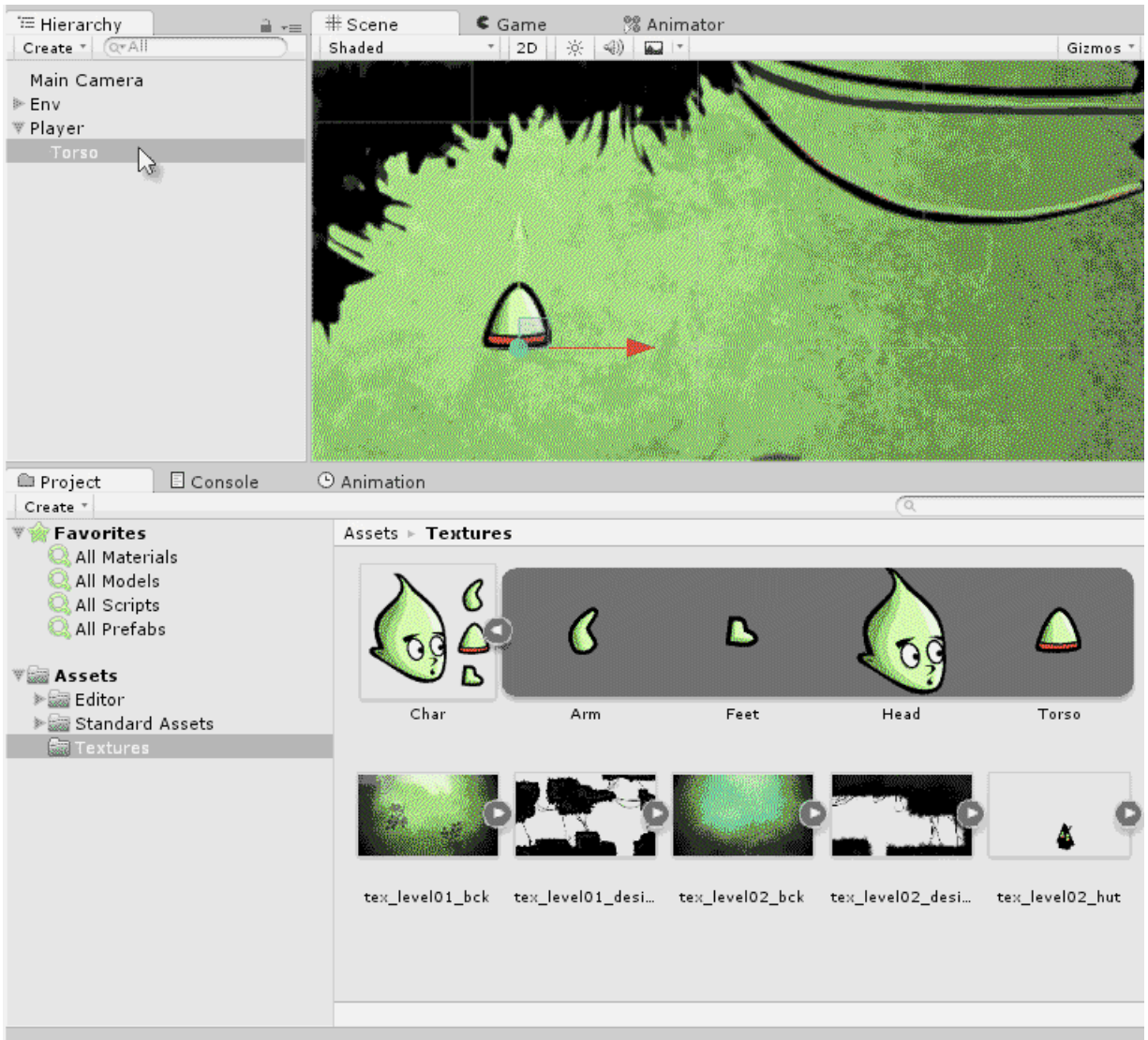


图5.36 开始设计一个玩家角色

在成功添加了玩家角色的躯干之后，我们接着向里面添加手臂和腿。手臂应该被添加为躯干的子对象，因为躯干决定了手臂的位置。不过，腿却应该被添加为玩家角色的子对象，这样腿和躯干是平级的关系，这是因为躯干的运动可以与腿不相干。图5.37中给出了完整的层次安排。每当添加一个肢体后，需要对其进行移动，使之出现在正确的位置。例如头应该位于脚的上方等。



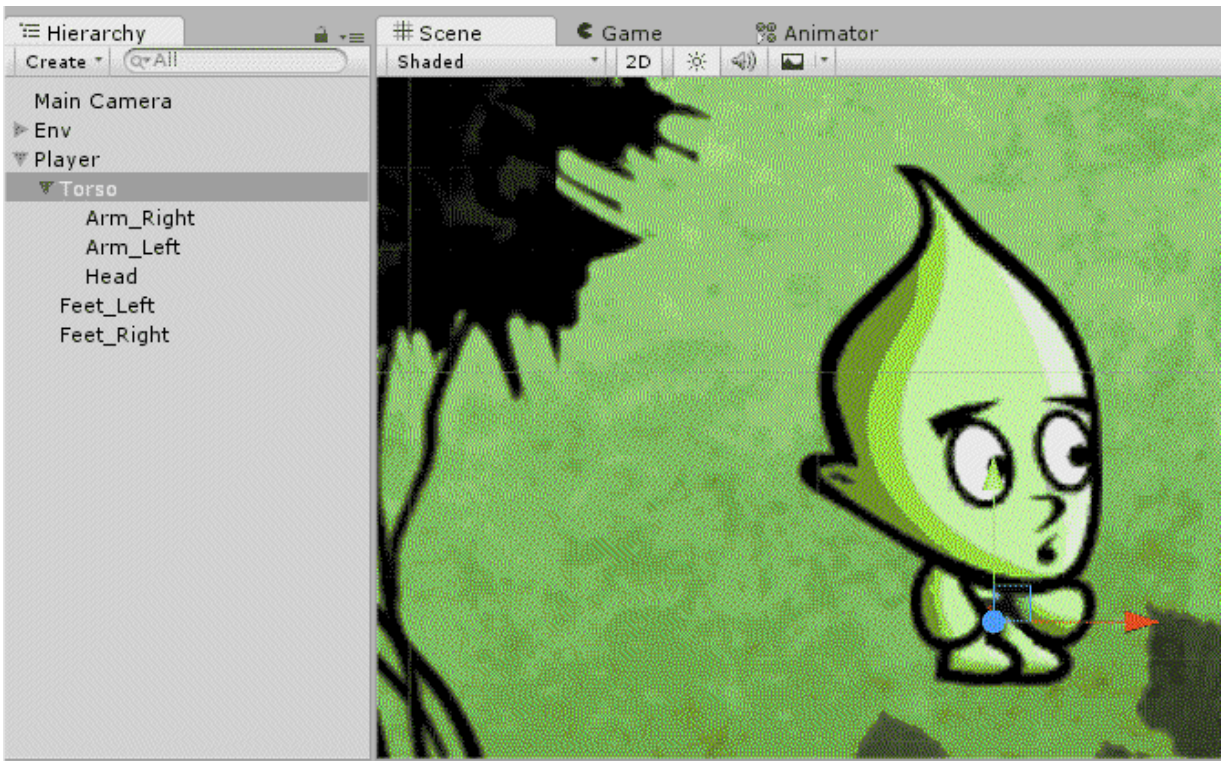


图5.37 构建一个游戏角色

默认情况下，身体各个部分的渲染顺序可能是不正确的，这是由于它们在“Sprite Renderer”组件中的序号都是相同的。这也就意味着Unity可能按照任意的顺序来对每个部分进行渲染，例如允许手臂出现在头部的前面，腿出现在身体的前面等。为了修正这个问题，我们将依次选择每个部分并为其指定适当的顺序值，要注意的是每个部分的顺序值要大于背景的顺序值，小于前景的顺序值。这里为身体分配的顺序值是103，头的顺序值是105，左手臂为102，右手臂为104，左腿为100，右腿为101，如图5.38所示。

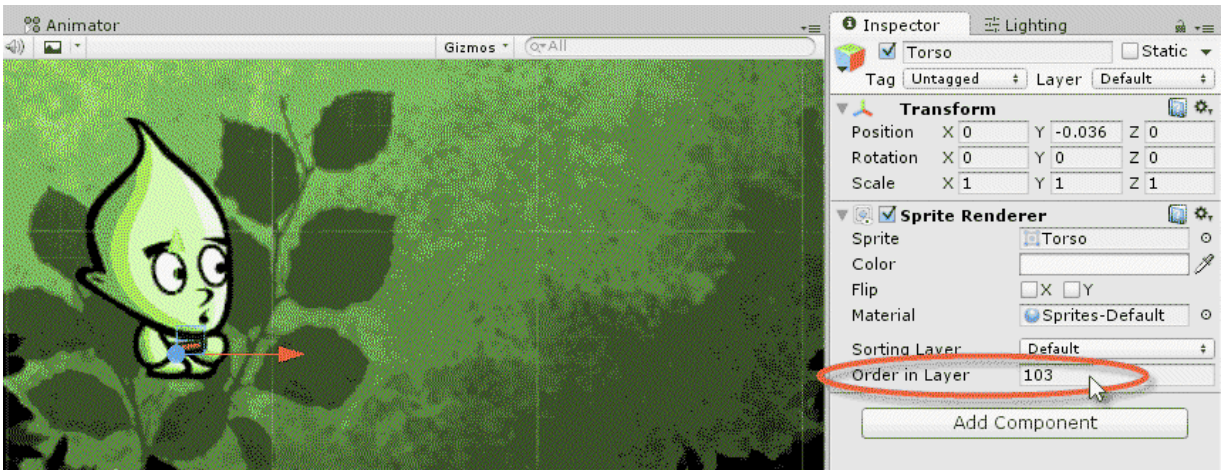


图5.38 身体部分的顺序值

现在已经成功地配置了肢体的渲染顺序。接下来是为玩家角色设置碰撞和物理属性。为了实现这些属性，需要向角色添加两个碰撞体，一个圆形碰撞体用来近似模拟玩家角色的脚，利用它就可以判断角色何时与地面发生了接触；一个盒子碰撞体用来近似模拟包括头部在内的整个身体。首先选中**Player**对象（位于最顶端的对象），然后依次单击“**Component | Physics 2D | Circle Collider 2D**”和“**Component | Physics 2D | Box Collider 2D**”，添加完的效果如图5.39所示。



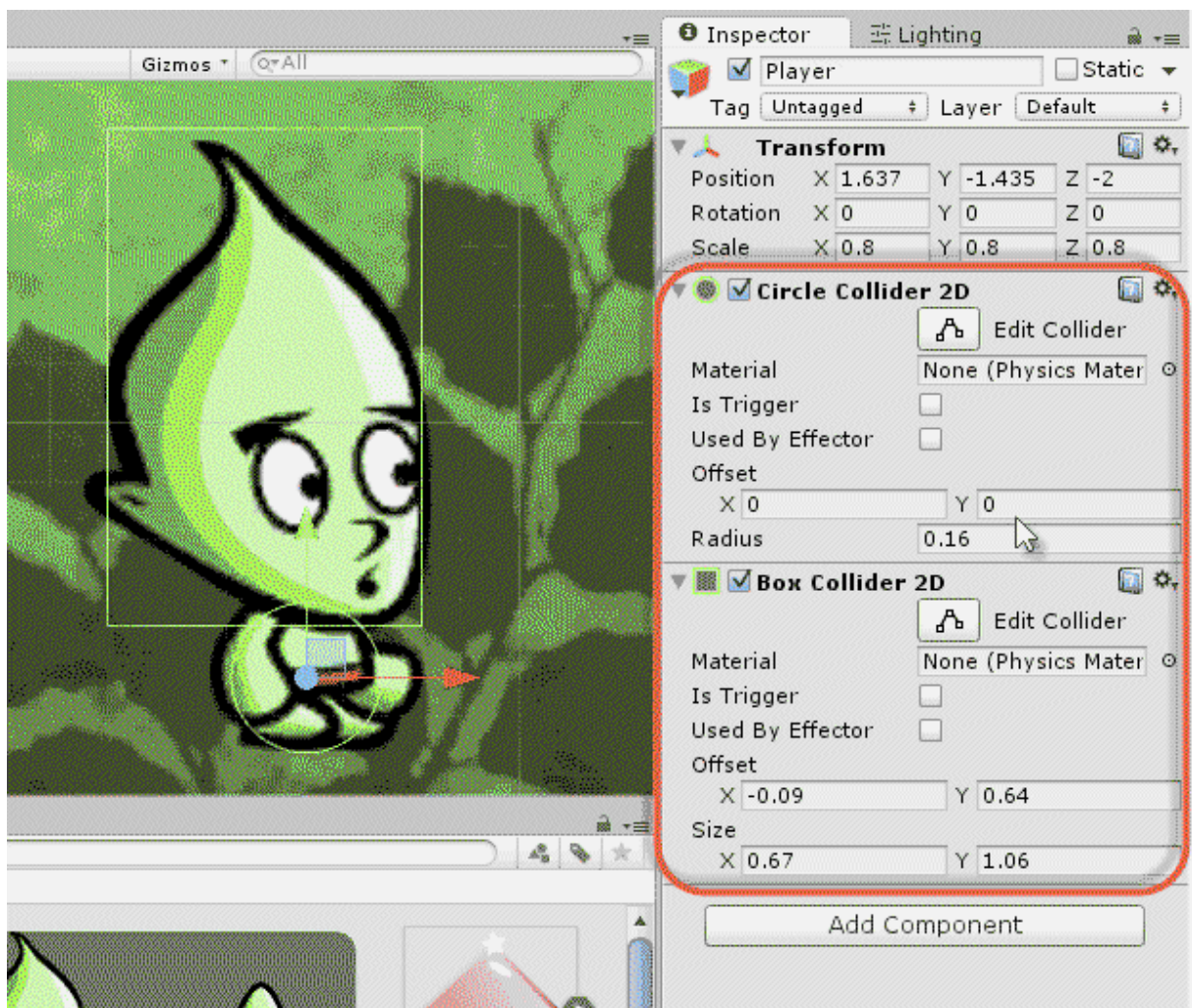


图5.39 向玩家对象添加两个碰撞体——圆形碰撞体和盒子碰撞体

圆形碰撞体是特别重要的，这是因为它是用来检测角色是否接触地面的主要手段，也可以用来检测角色移动中与地面的接触。基于这个原因，应该给这个圆形碰撞体分配一个物理材质来保证当它在场景中移动时不会产生摩擦而阻止角色的运动。为了实现这个功能，需要创建一个新的物理材质，首先在项目（Project）面板的空白处单击鼠标右键，然后在弹出的上下文菜单中依次选择“Create | Physics2D Material”，为这个材质命名为“Low Friction”，如图5.40所示。

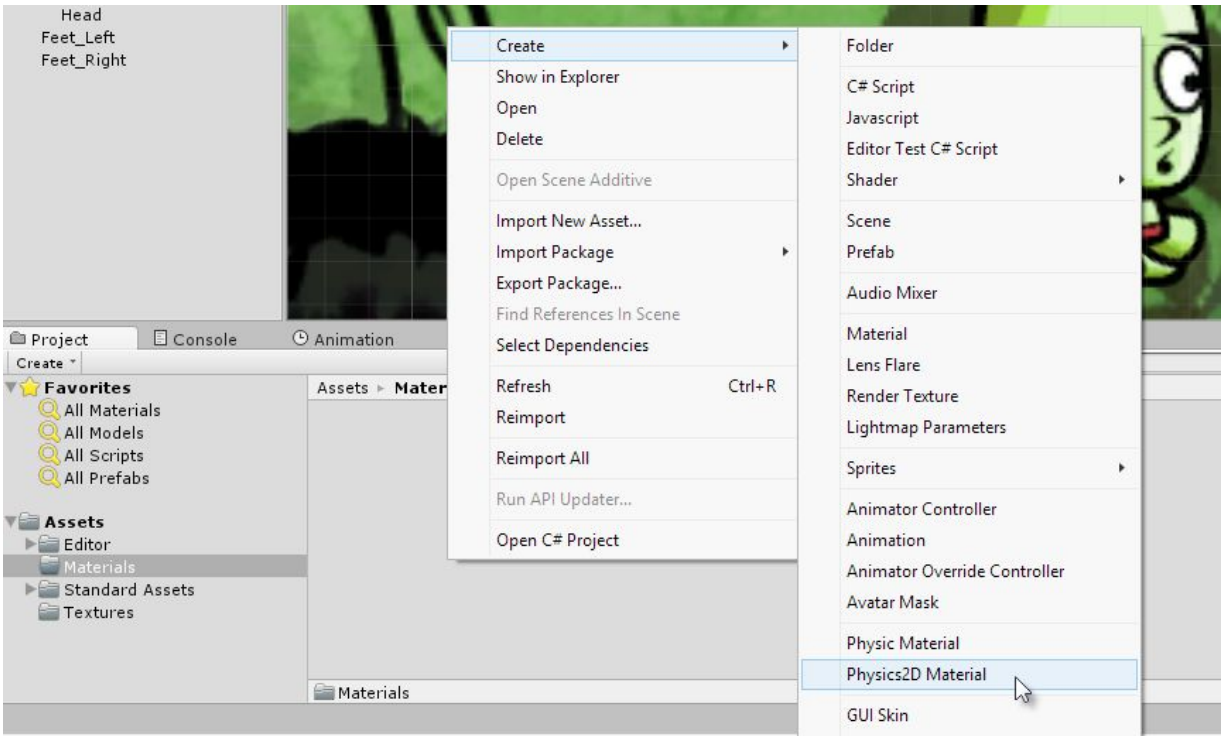


图5.40 创建一个新的物理材质

在项目（**Project**）面板上选中新创建的“**Physics2D**”材质，然后在对象检查（**Inspector**）面板上将它的**Friction**设置为0.1。之后从项目（**Project**）面板上将这个“**Physics2D**”材质拖曳到**Player**对象的**CircleCollider2D**组件中的**Material**后面的位置，如图5.41所示。利用这些设置，玩家角色就可以表现得更为真实。

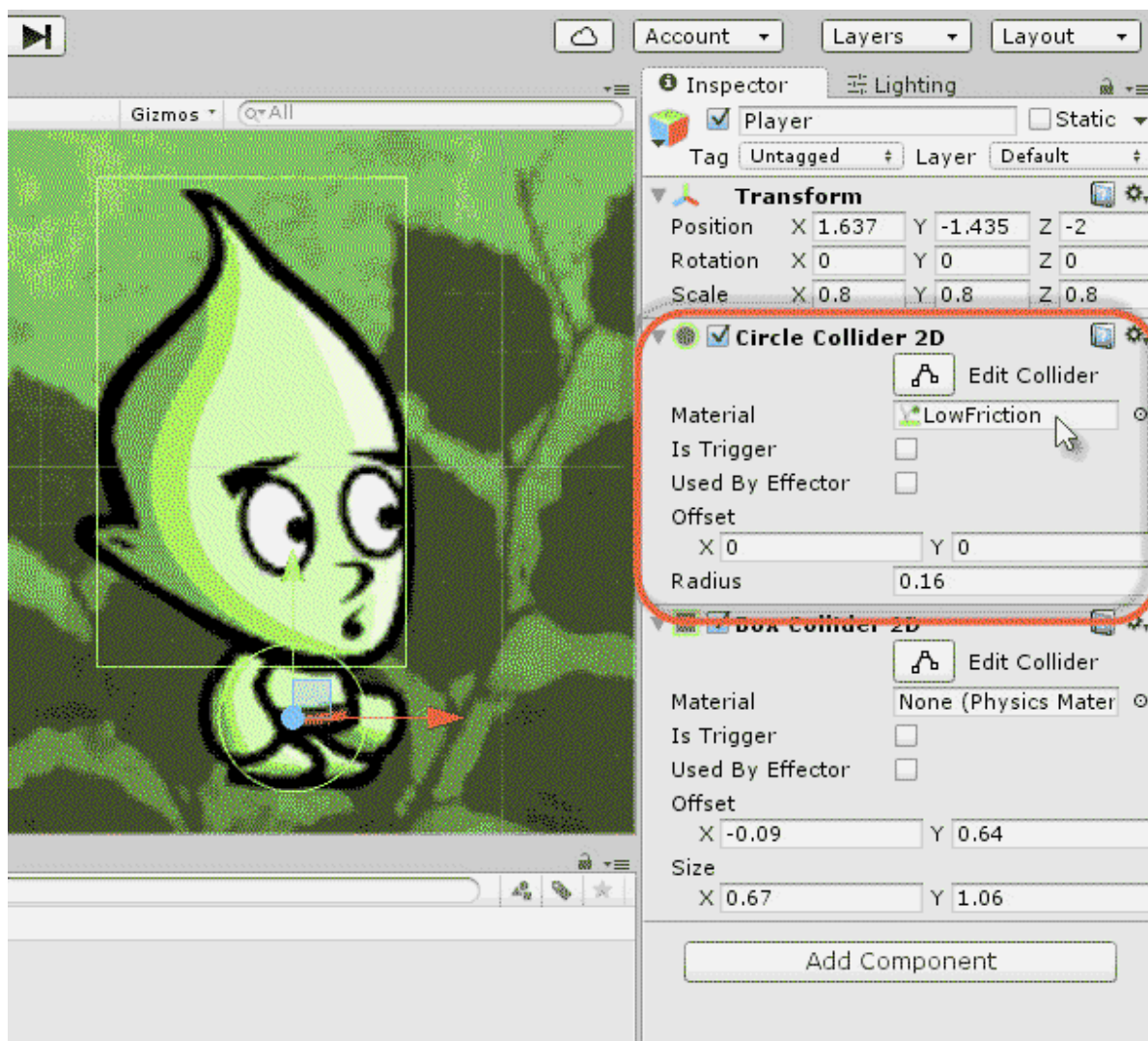


图5.41 为玩家角色分配一个物理材质

最后给Player对象分配一个“Rigidbody2D”，并将“Linear Drag”和“Gravity Scale”的值都设置为3。此外，将“Collision Detection”的值改变为“Continuous”，这样可以获得最精确的碰撞检测，然后将“Freeze Rotation”后面的Z复选框选中，这是因为玩家角色不应该转动。现在已经拥有了一个具备完整物理属性的玩家角色，如图5.42所示。



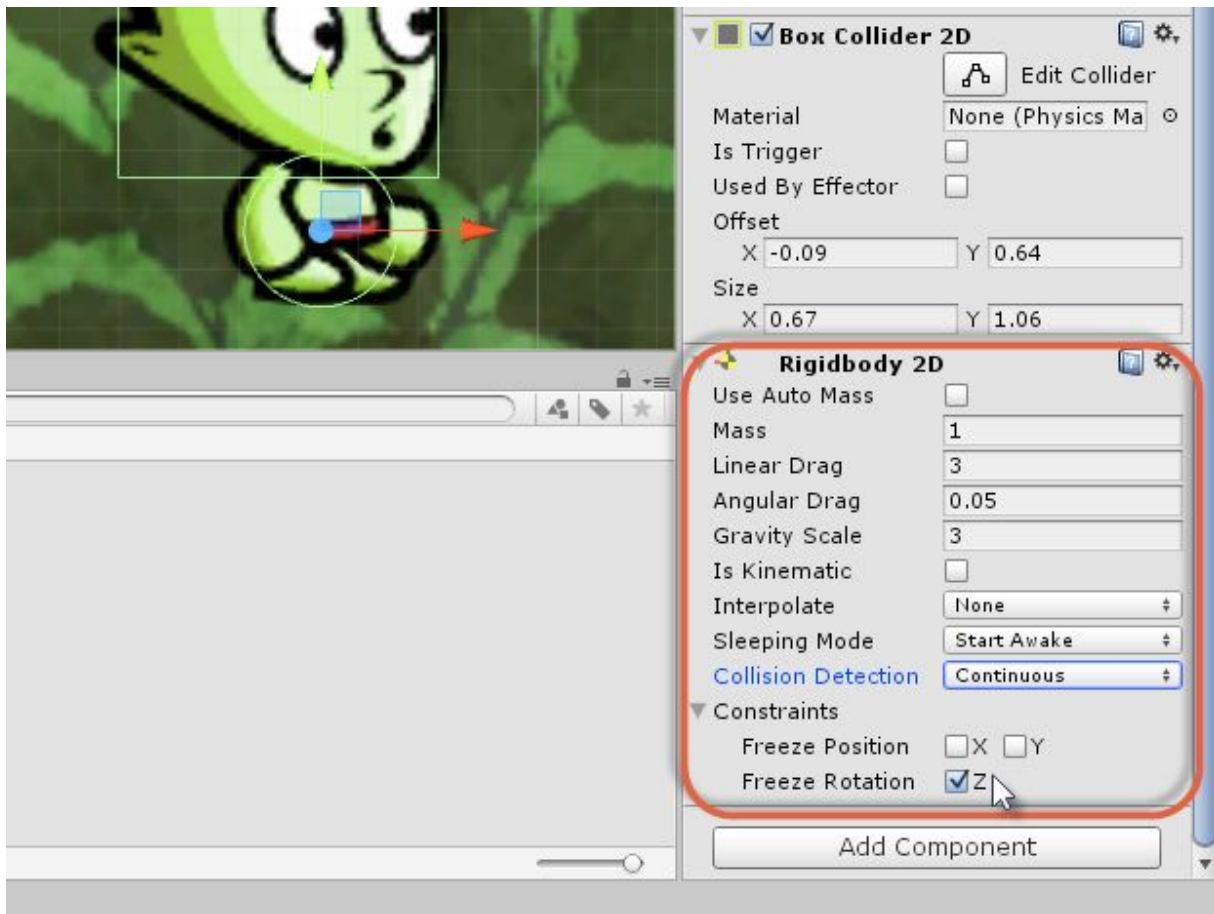


图5.42 为玩家角色设置物理属性

## 5.6 编写控制玩家移动的脚本

现在这个游戏已经包含了一个拥有碰撞属性的环境，一个由多个部分组成的可以和环境进行交互的玩家对象。然而，到现在为止玩家角色仍然是无法控制的，本节将进一步开发控制器的功能。用户将使用两个主要的输入机制，就是运动（向左或者向右行走）和跳跃。这个输入可以轻松地使用CrossPlatformInputManager包来读取，这个包是Unity自带的资源包。这个包在项目创建的时候就已经导入了，不过现在一样可以通过在应用程序菜单上依次单击“Assets | Import Package |

CrossPlatformInput”来导入。完成了导入操作之后，打开“Assets | CrossPlatformInput | Prefabs”文件夹，然后将“MobileTiltControlRig prefab”拖曳到场景中。前面的章节涉及过这个预设体，它可以读取一些设备上的输入数据，并将其映射到水平和垂直轴线，如图5.43所示。

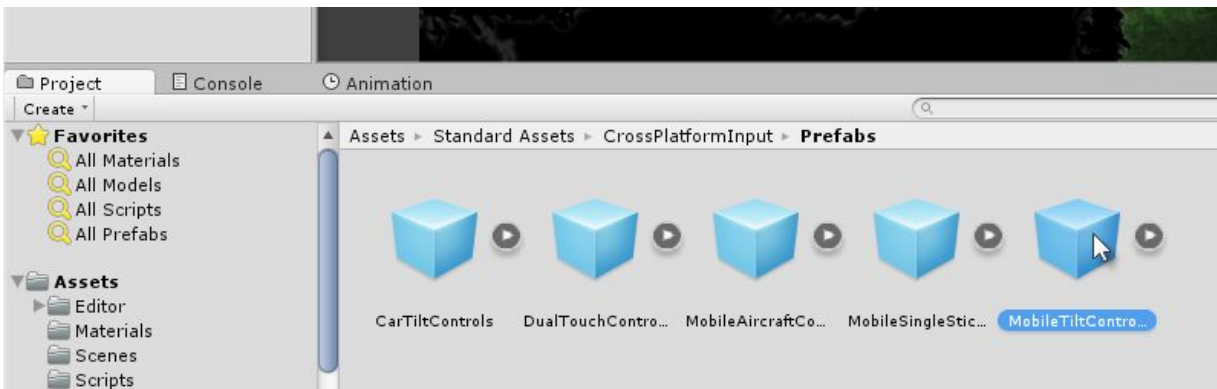


图5.43 跨平台输入预设体实现了方便的多设备控制

现在来编写玩家角色控制的脚本。首先创建一个名为“PlayerControl.cs”的新脚本，然后将这个脚本附加到Player角色上。下面的代码示例5.1给出了这个脚本的完整代码。

### 代码示例5.1:

```
//-----  
using UnityEngine;  
using System.Collections;  
using UnityStandardAssets.CrossPlatformInput;  
//-----  
public class PlayerControl : MonoBehaviour  
{  
    //-----  
    public enum FACEDIRECTION {FACELEFT = -1, FACERIGHT = 1};  
    //Player对象面对的方向是左还是右?  
    public FACEDIRECTION Facing = FACEDIRECTION.FACERIGHT;  
    //哪些对象的tag被标记为ground  
    public LayerMask GroundLayer;  
    //对刚体的引用  
    private Rigidbody2D ThisBody = null;
```

```

//对transform组件的引用
private Transform ThisTransform = null;
//对脚部碰撞体的引用
public CircleCollider2D FeetCollider = null;
//我们碰到地面了吗?
public bool isGrounded = false;
//主要输入轴
public string HorzAxis = "Horizontal";
public string JumpButton = "Jump";
//速度变量
public float MaxSpeed = 50f;
public float JumpPower = 600;
public float JumpTimeOut = 1f;
//我们现在可以跳了吗
private bool CanJump = true;
//我们可以控制游戏角色了吗?
public bool CanControl = true;
public static PlayerControl PlayerInstance = null;
//-----
public static float Health
{
    get
    {
        return _Health;
    }

    set
    {
        _Health = value;

        //如果死亡的话，游戏也就结束了
        if(_Health <= 0)
        {
            Die();
        }
    }
}

[SerializeField]
private static float _Health = 100f;
//-----
// 初始化
void Awake ()
{
    //获取transform组件和 rigid body组件
    ThisBody = GetComponent<Rigidbody2D>();
    ThisTransform = GetComponent<Transform>();

    //设置静态实例
    PlayerInstance = this;
}

```



```

//-----
//返回一个布尔值，玩家在地面上吗？
private bool GetGrounded()
{
    //检测地面
    Vector2 CircleCenter = new Vector2(ThisTransform.position.x,
        ThisTransform.position.y) + FeetCollider.offset;
    Collider2D[] HitColliders =
        Physics2D.OverlapCircleAll(CircleCenter,
            FeetCollider.radius, GroundLayer);
    if(HitColliders.Length > 0) return true;
    return false;
}
//-----
//调转角色方向
private void FlipDirection()
{
    Facing = (FACEDIRECTION) ((int)Facing * -1f);
    Vector3 LocalScale = ThisTransform.localScale;
    LocalScale.x *= -1f;
    ThisTransform.localScale = LocalScale;
}
//-----
//跳跃
private void Jump()
{
    //如果我们在地面上的话，就跳跃
    if(!isGrounded || !CanJump)return;

    //跳跃
    ThisBody.AddForce(Vector2.up * JumpPower);
    CanJump = false;
    Invoke ("ActivateJump", JumpTimeOut);
}
//-----
//在指定时间结束后激活允许跳跃变量
//禁止第一次跳跃未结束时就二次起跳
private void ActivateJump()
{
    CanJump = true;
}
//-----
// Update函数在每一帧调用一次
void FixedUpdate ()
{
    //如果我们不能控制角色，就退出
    if(!CanControl || Health <= 0f)
    {
        return;
    }
}

```

```

//更新 grounded 变量状态
isGrounded = GetGrounded();
float Horz = CrossPlatformInputManager.GetAxis(HorzAxis);
ThisBody.AddForce(Vector2.right * Horz * MaxSpeed);

if(CrossPlatformInputManager.GetButton(JumpButton))
    Jump();

//对速度进行限制
ThisBody.velocity = new
    Vector2(Mathf.Clamp(ThisBody.velocity.x, -MaxSpeed,
        MaxSpeed),
        Mathf.Clamp(ThisBody.velocity.y, -Mathf.Infinity,
            JumpPower));

//如果需要的话就调转方向
if((Horz < 0f && Facing != FACEDIRECTION.FACELEFT) ||
    (Horz > 0f && Facing != FACEDIRECTION.FACERIGHT))
    FlipDirection();
}
//-----
void OnDestroy()
{
    PlayerInstance = null;
}
//-----
//杀死玩家的功能
static void Die()
{
    Destroy(PlayerControl.PlayerInstance.gameObject);
}
//-----
//重置Player
public static void Reset()
{
    Health = 100f;
}
//-----
}
//-----

```

下面对代码示例5.1进行总结。

- **PlayerControl**类负责处理所有的玩家输入，并控制玩家的左右运动以及跳跃。

- 为了实现玩家的运动，在**ThisBody**变量中存在一个对**Rigidbody2D**组件的引用，这个**ThisBody**变量在**Awake**函数中赋值。玩家的运动和动作使用**Rigidbody2D.Velocity**变量实现。关于这个变量的更多信息可以访问在线文档<http://docs.Unity3d.com/ScriptReference/Rigidbody2D-velocity.html>。
- 函数**FlipDirection**用来将图像精灵的水平值进行反转，转头面向左或者右（改变图像的方向，例如1和-1）。从Unity 5.3版本起，可以使用**SpriteRenderer**组件的**Flip**属性来代替。
- **FixedUpdate**函数用来代替**Update**函数来实现对**Player**角色的更新操作，这是因为正在使用**Rigidbody2D**，也就是一个基于物理属性的组件。所有的物理功能应该在**FixedUpdate**函数中调用，调用的时间间隔应该是固定的，所以时间单位应该是每秒而不是每帧。关于它的更多详细信息可以在Unity的在线文档<http://docs.Unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html>中找到。
- 函数**GetGrounded**会检测场景中特定层中的其他碰撞体与原型碰撞体之间的交叉和重叠，简而言之，这个函数会给出当前玩家角色的脚是否与地面相接触。如果是，玩家就可以执行跳跃操作；否则，玩家不可以跳跃。因为他们现在已经在空中了，凌空再跳跃这种操作是不允许的。

为了让前面的代码能够正常地工作，必须对玩家角色和场景做出一些细微的调整。特别是**GetGrounded**函数需要将同一层中的地面都集中到同一层。这也就意味着关卡前景应该与其他对象位于不同的层上。首先创建一个名为“**Ground**”的新层，然后将前景对象分配到这一层中。创建的方法是，首先选中前景对象，然后在对象**Inspector**面板上，单击

Layer下拉列表框，然后在上下文菜单中选中“Add Layer”，如图5.44所示。

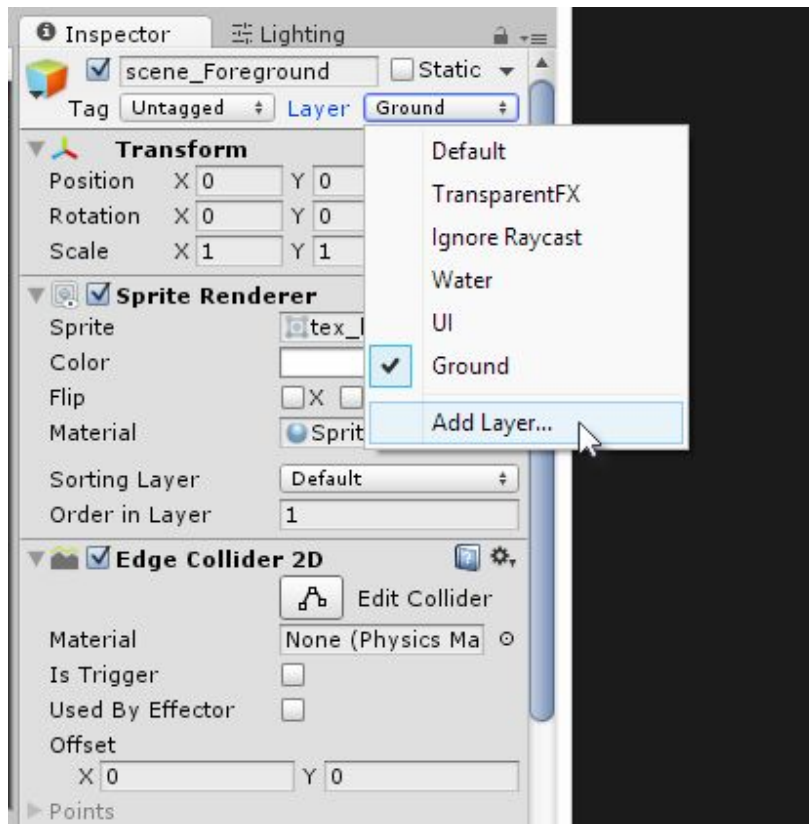


图5.44 添加一个新的层

现在添加一个名为“Ground”的新层，只需在一个可用的字段区域输入“Ground”，如图5.45所示。

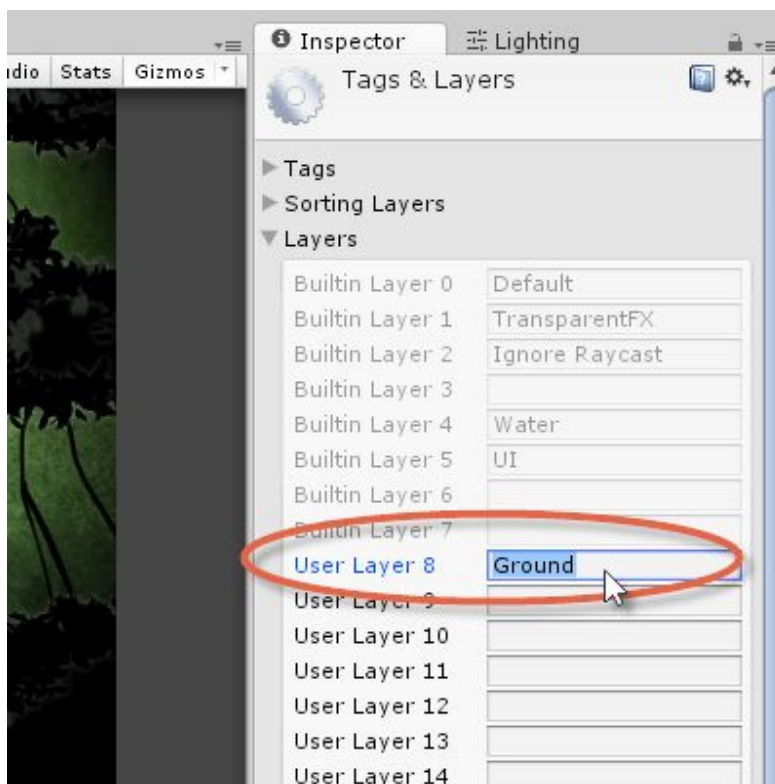


图5.45 创建一个新的“Ground”层

现在，将前景对象分配到Ground层。首先选中前景对象，然后在对象Inspector面板中选中layer下拉列表框中的“Ground”选项。在将前景对象分配到Ground层之后，脚本PlayerControl要求必须标明哪一层被指定为Ground。首先选中Player对象，然后在对象检查（Inspector）面板中将“Ground Layer”的值设置为“Ground”，如图5.46所示。

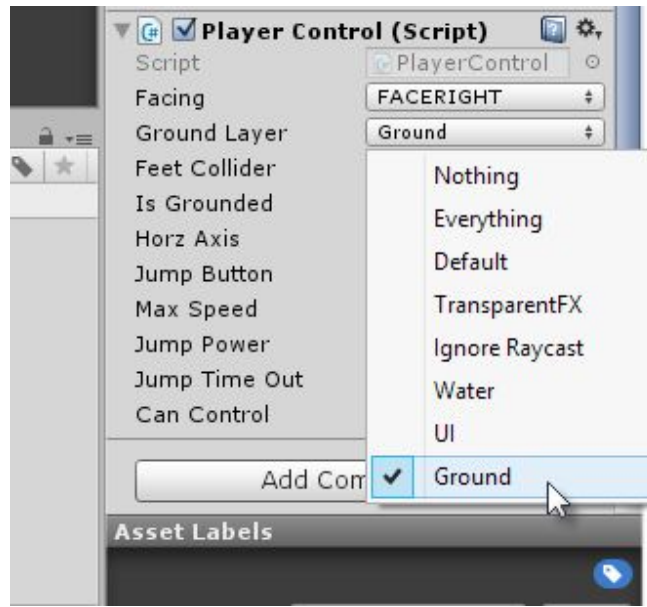


图5.46 为碰撞检测设置Ground层

另外，脚部碰撞体（Feet Collider）区域的值也需要制定，以指明使用哪个碰撞体对象来进行地面碰撞的检测。对于这个区域，我们需要将圆形（Circle Collider）组件拖动到“Feet Collider”区域，如图5.47所示。



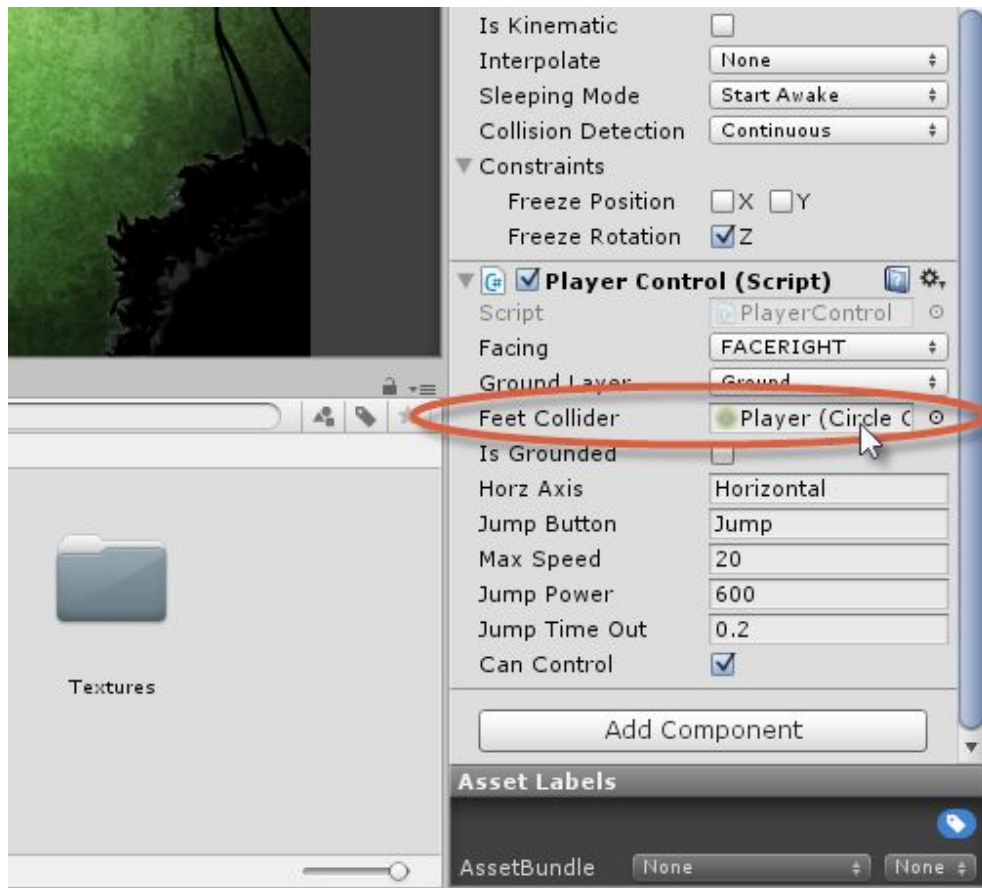


图5.47 脚部碰撞体用来检测玩家角色何时与地面接触

现在来对玩家角色进行一下测试，首先在工具栏上单击“Play”图标，测试玩家角色的控制。使用W、A、S、D（或者方向键）可以控制玩家角色的移动。空格键就可以控制角色进行跳跃，如图5.48所示。



图5.48 对玩家角色进行测试

## 5.7 优化

到目前为止，已经开发了一个有趣的环境和一个可控的角色。在继续游戏的开发之前，将注意力转移到优化方面，这是一个在开发过程必须要考虑的问题。优化指的是可以应用提高运行性能和改善工作流程的技巧。现在考虑使用预设体来改善工作流程，使用图像精灵打包技术来提高运行时的性能，现在就从预设体开始。

预设体（Prefab）是一个Unity中的资源，利用预设体就可以将场景中的许多对象集合到一起并将它们封装成一个单独的单元。预设体可以被当作一个资源添加到项目（Project）面板上。从这里开始，预设体就可以作为一个完整的单元添加到其他的场景或者环境。玩家角色就是一

个理想的预设体候选，因为它必须应用到所有的场景中。从玩家开始创建预设体，只需简单将Player对象拖曳到项目（Project）面板中一个名为Prefabs的独立文件夹中，如图5.49所示。

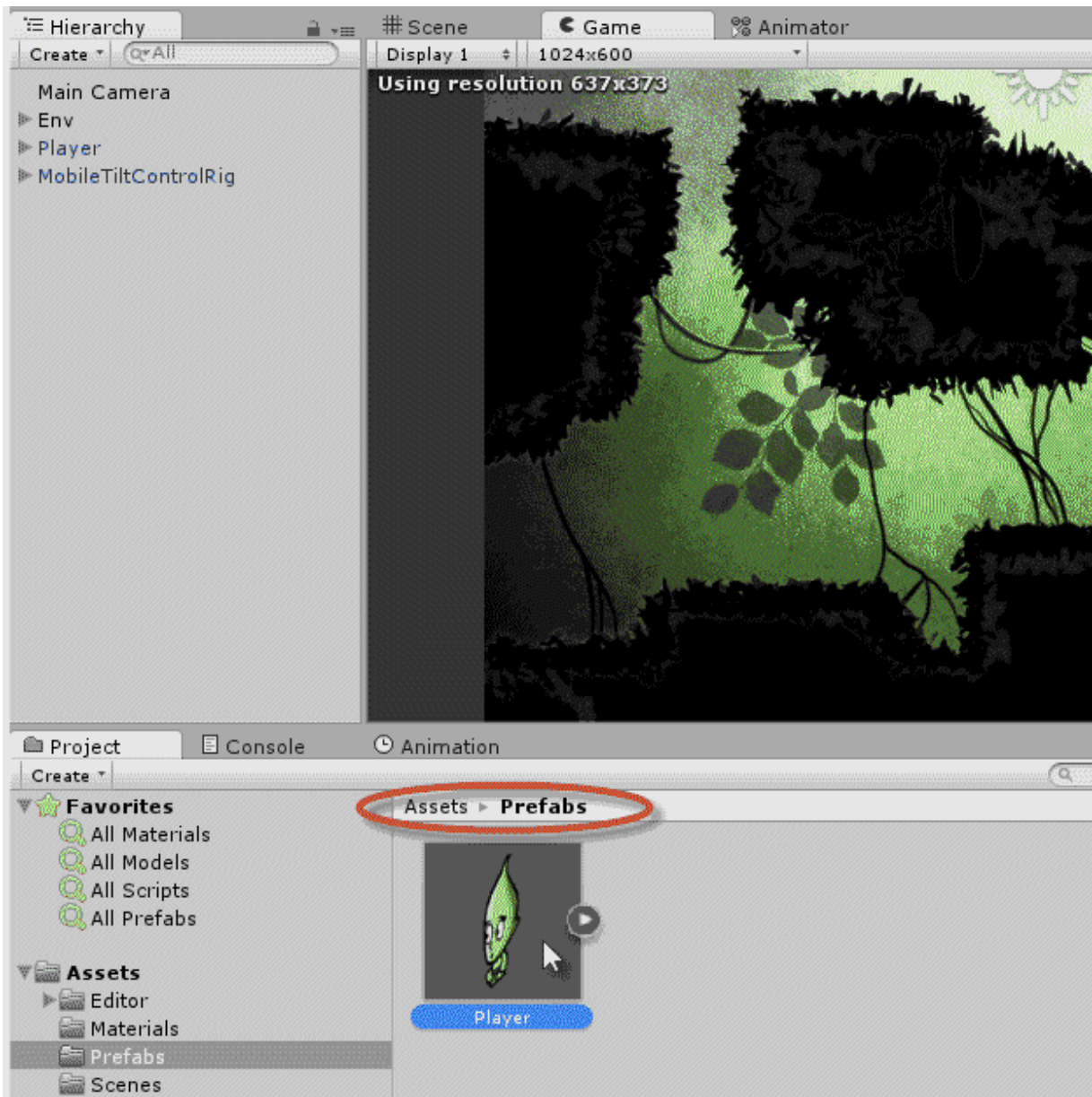


图5.49 创建一个玩家预设体

在创建了预设体之后，层次（**Hierarchy**）面板上的**Player**对象的名字就变成了蓝色，这表明它已经关联到了预设体资源上。当在项目（**Project**）面板上选中预设体以后，如果在对象检查（**Inspector**）面板中对其进行了修改，那么场景中的玩家（**Player**）就会相应地做出改变。不过也可以切断场景中的玩家（**Player**）和预设体之间的这种关联，切断的方法就是选中玩家（**Player**）之后，在应用程序菜单中依次选中“**GameObject | Break Prefab Instance**”。这样就将场景中的对象转换成为一个预设体的独立副本，如图5.50所示。



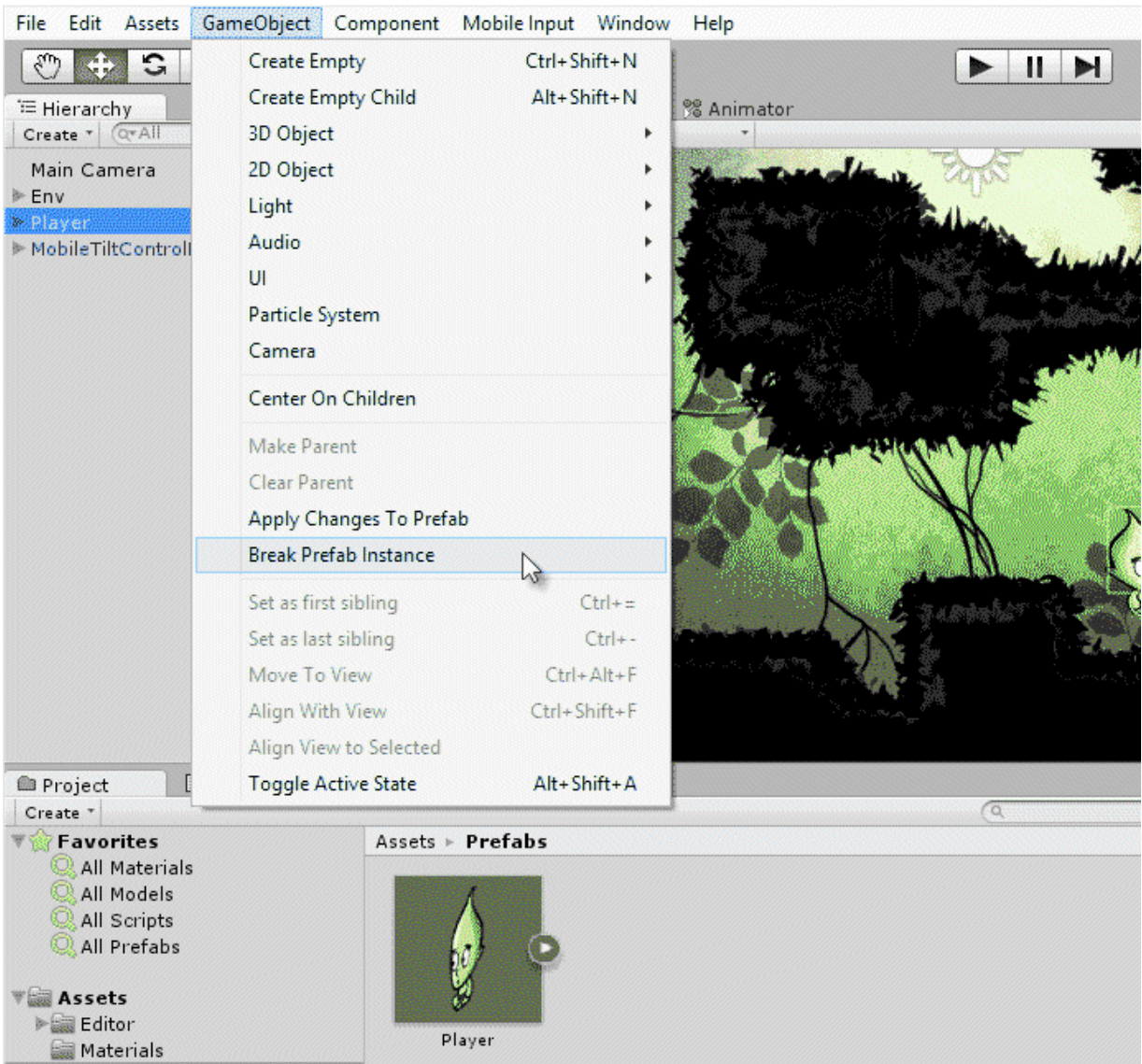


图5.50 切断预设体和实例之间的联系

大多数时候希望保持实例对象和预设体之间的关联。有时，可能对场景中的对象做出了一些修改，之后又希望将这些修改再应用到项目（Project）面板的预设体资源上，影响到所有其他相关联的实例对象。如果想实现这个功能，需要首先选中作出了修改的那个对象，然后从应用程序菜单处依次选中“GameObject | Apply Changes to Prefab”，如图5.51所示。

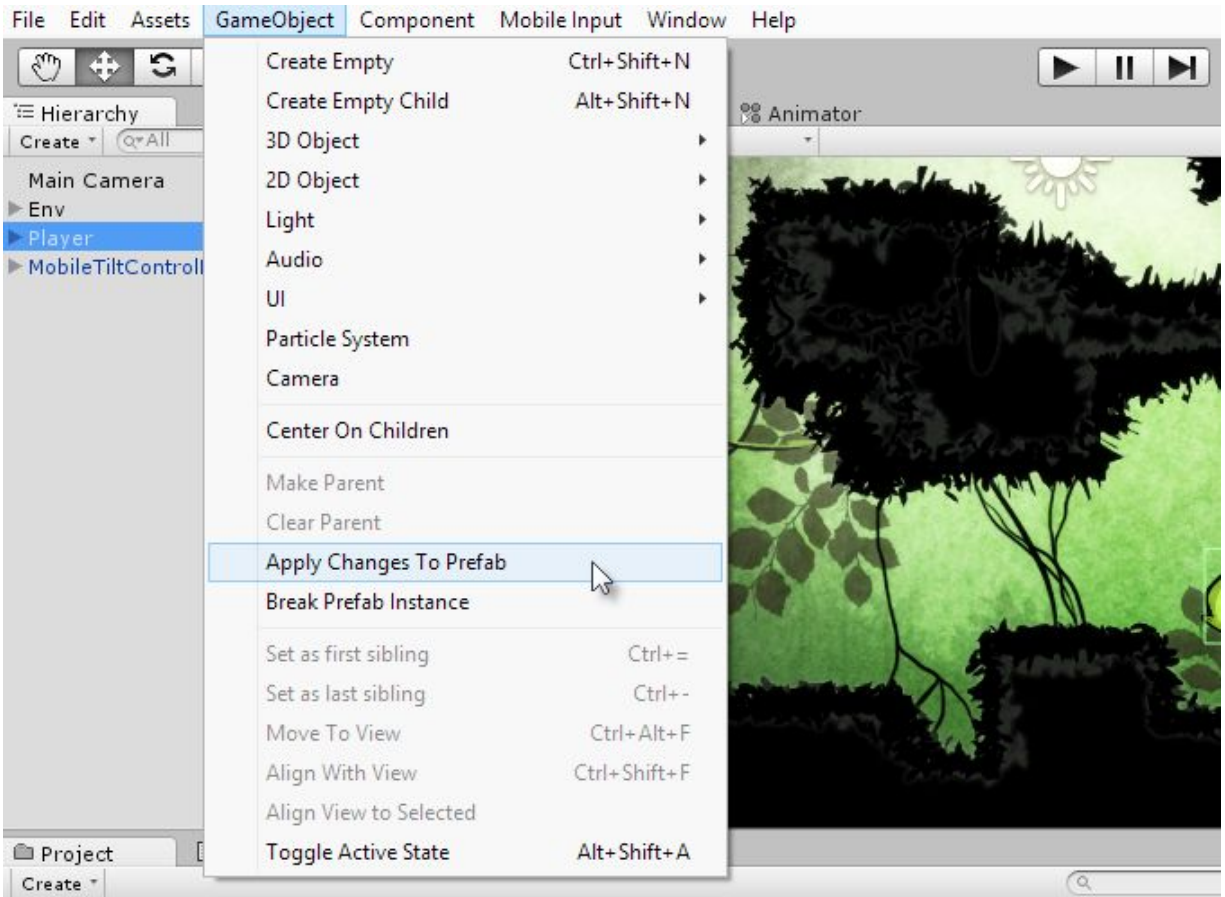


图5.51 将改变应用到预设体

除了使用预设体之外，还需要对2D游戏进行渲染，对性能进行优化。当运行游戏时，Unity会对当时屏幕中出现的每一个独立的贴图或者Sprite进行独立唯一的绘制调用（Draw Call）。绘制调用指的就是Unity在将一个网格、材质或者贴图显示在屏幕上时必须执行的一个步骤或者过程周期。“Draw Call”往往也就意味着系统的开销，所以最好尽可能地减少绘制调用的使用。

对2D游戏来说，可以通过将相关的贴图，例如场景中的所有道具、所有的敌人、所有的武器进行组合处理。这就是说，通过将一组贴图一起交给Unity处理，Unity就可以实现内部优化，从而提高渲染性能。具



体来说，Unity会使用一个单独的大型贴图来代替那些相关的贴图。如果想要实现这个优化，需要首先选中所有的道具贴图，对于这个游戏来说，道具需要包含“Player”、房子、平台以及宝石。这些贴图都包含在项目（Project）面板，虽然在游戏中可能还没使用到。选中这些贴图，在对象检查（Inspector）面板中为它们在“Packing Tag”处起一个相同的名字（Props），然后单击“Apply”按钮，如图5.52所示。

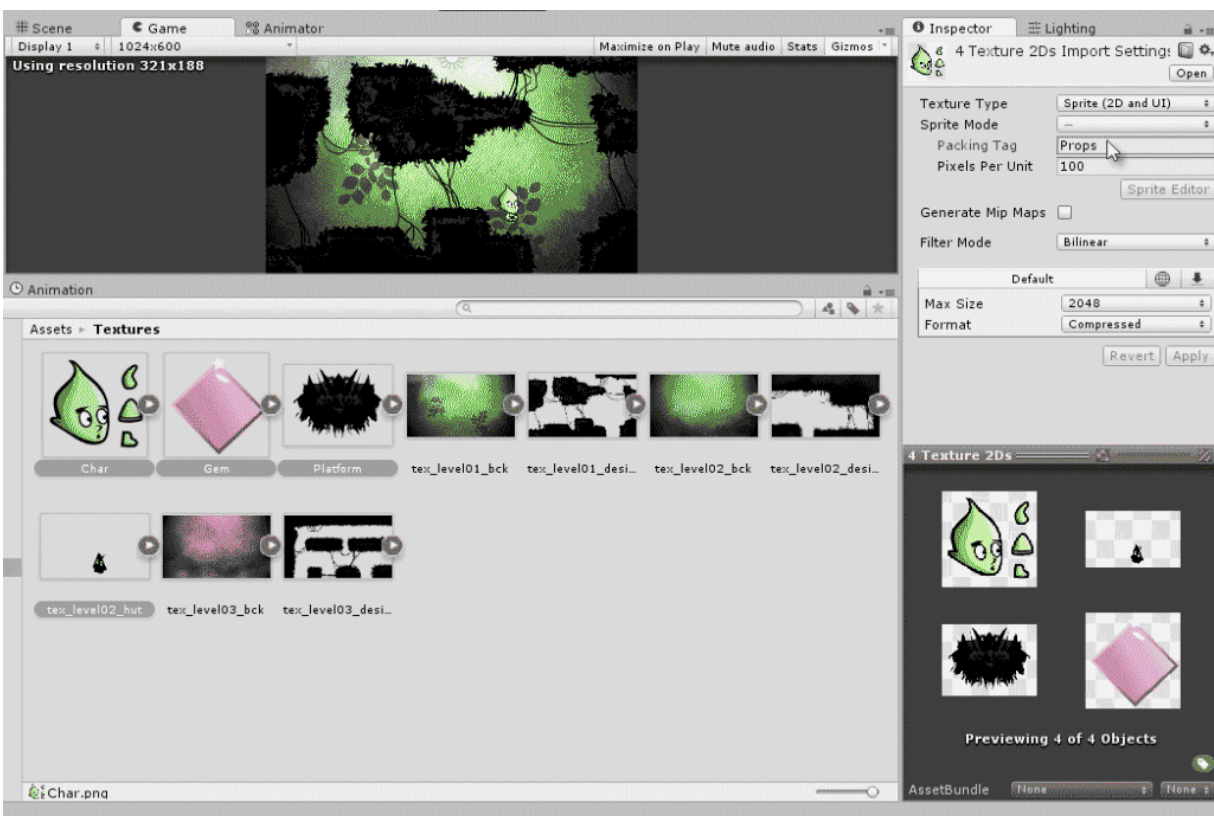


图5.52 为多个贴图设置相同的“Packing Tag”属性值

对背景重复这个过程，首先选中所有的背景，然后将它们的Packing Tag属性都设置为“Background”，然后单击“Apply”，如图5.53所示。

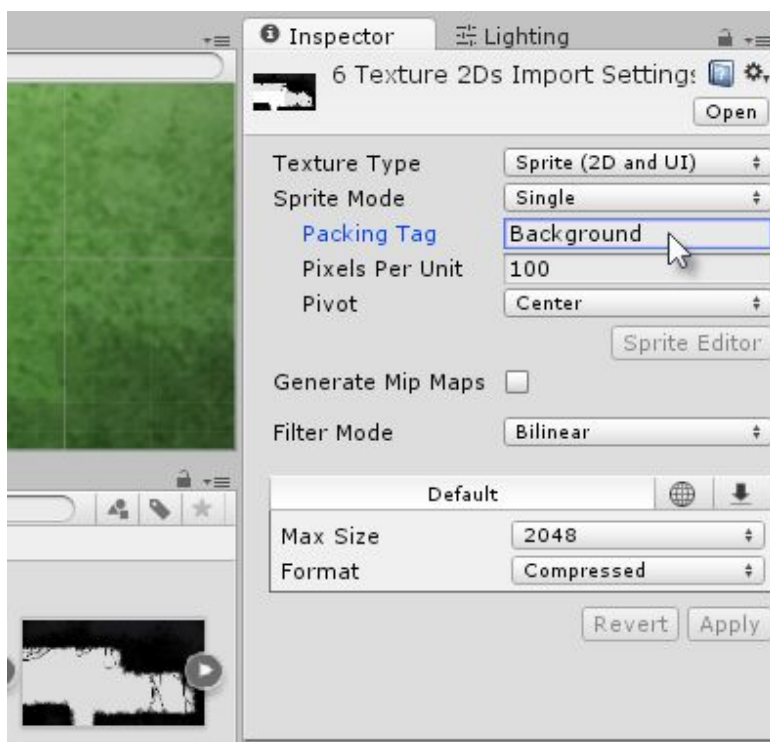


图5.53 创建一个背景贴图组合

按下Play按钮，Unity就会自动基于分组来批量处理贴图以实现性能的优化，这个技术可以明显减少绘制调用的使用次数。当按下Play按钮之后，Unity内部就会产生一个新的贴图集合，同时会出现一个进度条。在Play模式中，可以在“Sprite Packer”窗口中看到Unity是如何对贴图进行组织的。可以在应用程序菜单中依次选择“Window | Sprite Packer”，如图5.54所示。

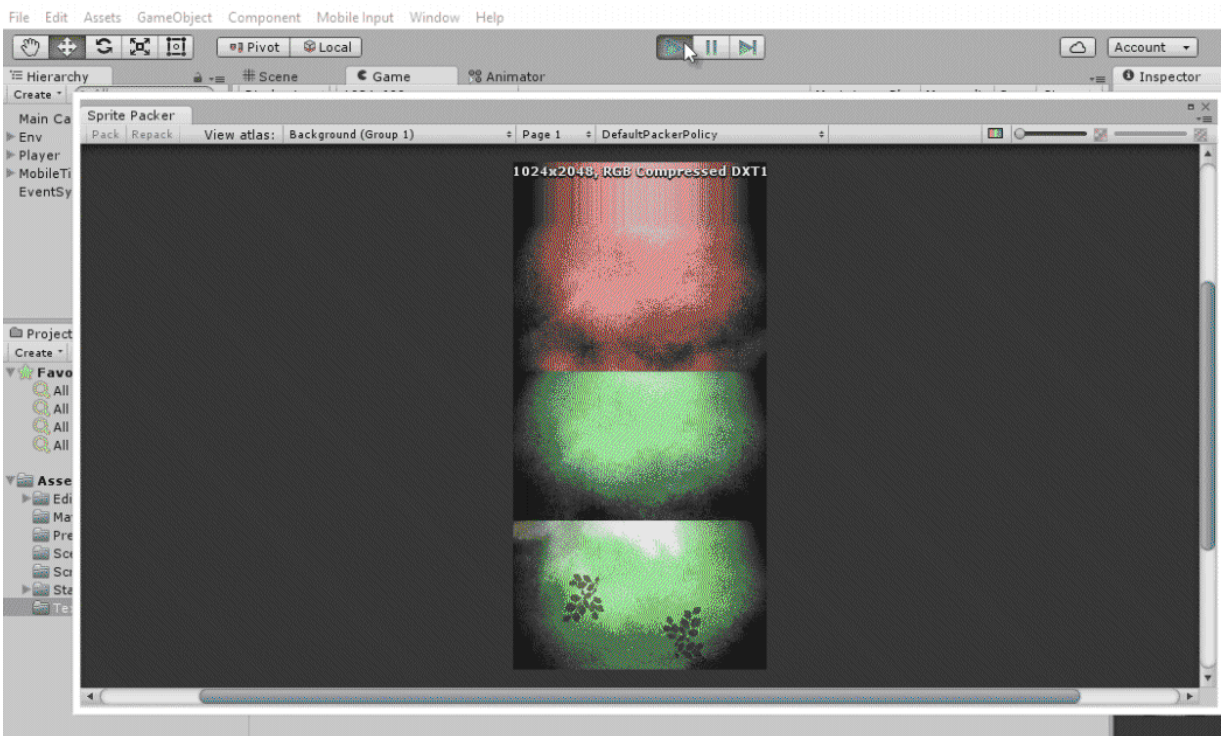


图5.54 Unity将所有标记相同的贴图组织成一个图集

## 5.8 小结

干得很出色！我们在本章中已经做了不少工作了，从最初的一个空无一物的项目，做成了现在这个很有意思的二维游戏，在这个游戏中玩家角色可以在一个具有物理属性的二维世界进行探索，而且这个角色可以左右移动、跳跃，图像精灵贴图也可以根据不同的运动做出变化。此外，我们也可以使用“**Sprite Packing**”技术对运行时的性能进行优化，这一点对于移动设备来说是十分理想的，在下一章中，我们将再接再厉，为游戏中添加更多的障碍和宝物等游戏物品。

## 第6章 二维冒险游戏（II）

上一章开始了二维冒险游戏的设计之旅。到现在为止，已经创建了一个受控制的角色，并且这个角色具有物理属性、碰撞检测以及重力功能，可以在关卡中行走。在这一章中，会将其余的功能添加到这个二维游戏项目中去，具体而言，本章将会涵盖以下主题：

- 移动障碍物例如升降平台
- 攻击玩家的炮塔
- 一个具有任务系统的非玩家控制角色（NPC）



本章一开始所使用的项目和资源可以在本书的配套文件中的 **Chapter06/Start** 文件夹中找到，如果没有完成之前的项目，那么现在就可以利用这个文件来开始本章的学习内容。

### 6.1 移动的平台

现在进一步将冒险元素添加到现有场景中来，具体来说，就是要添加一个移动的平台对象。这个平台应该向上移动，然后再向下，像个乒乓球一样在两端之间移动，并且不断地循环这个动作。游戏玩家可以跳到这个平台上，然后再移动到目标位置。这个平台对象可以被

构造成一个预设体，这样就可以在整个场景中反复使用，图6.1给出添加了一个这样移动平台的场景。



图6.1 创建一个移动的平台

首先在项目面板处选中平台的贴图，然后在对象检查（Inspector）面板上确认这个贴图已经被指定为图片精灵（Sprite）类型，而且它的“SpriteMode”属性已经被设置为“Single”，我们将平台贴图拖曳到场景中，然后将它的“Scale”属性设置为（0.7, 0.5, 1），如图6.2所示。



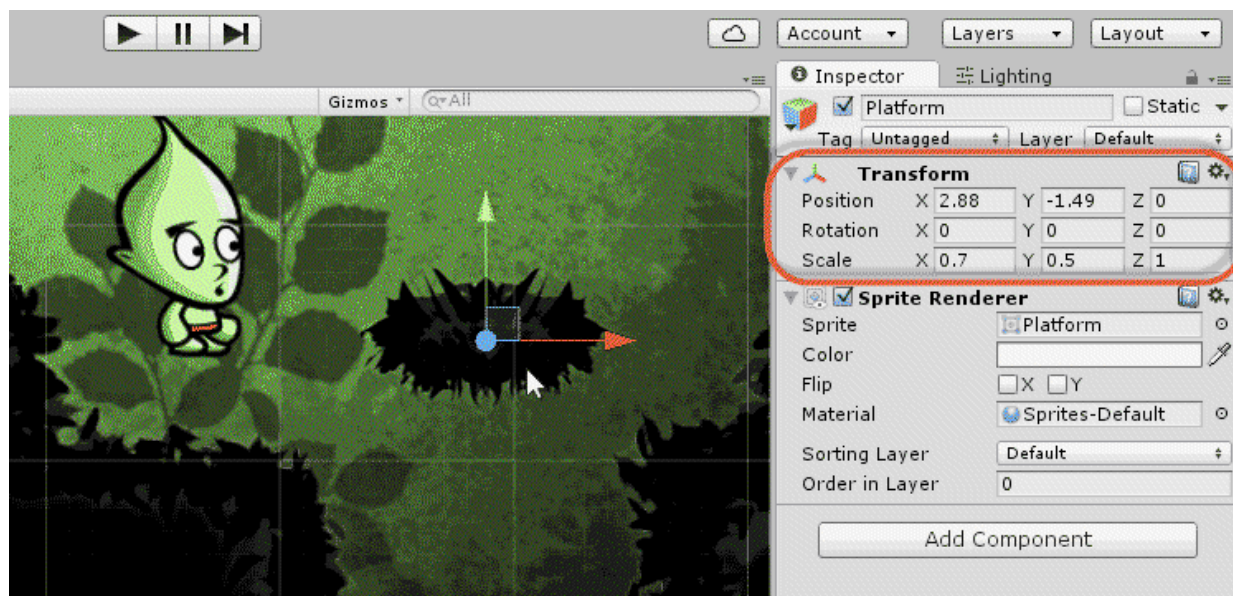


图6.2 构建一个可移动的平台

接下来，这个平台应该是一个实体对象，这样玩家才能和这个平台发生碰撞。要知道，我们的玩家角色应该可以站在这个平台上。因此，必须向这个平台对象上添加一个碰撞体。在这种情况下，二维盒子碰撞体（Box Collider 2D）是最为合适的。我们首先在场景选中平台对象，然后在菜单中依次选中“Component | Physics 2D | Box Collider 2D”，就可以为其添加一个二维盒子碰撞体（Box Collider 2D），如图6.3所示。



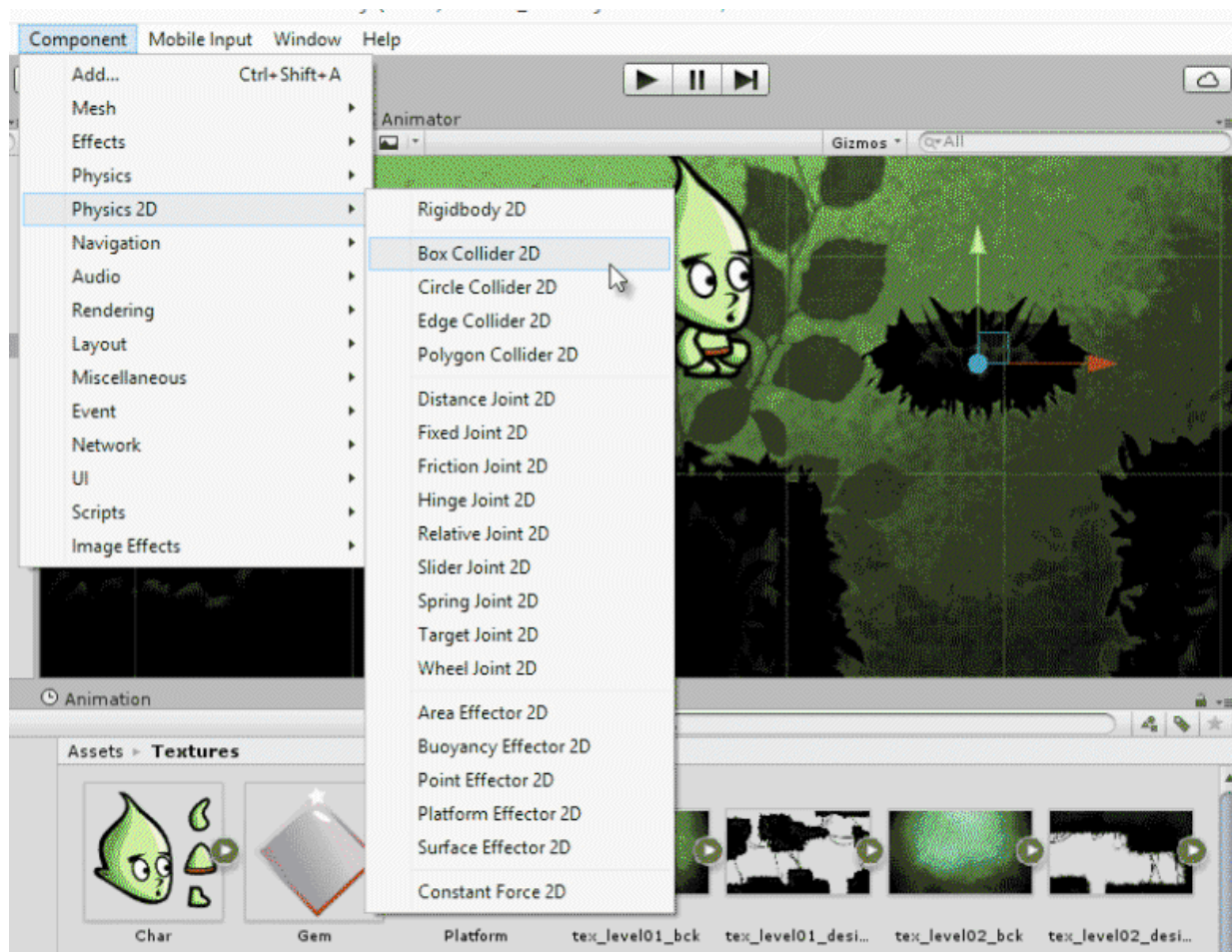


图6.3 向平台添加一个二维盒子碰撞体（Box Collider 2D）

当向这个平台添加了碰撞体（Collider）之后，就可以在对象检查（Inspector）面板中对这个碰撞体的属性进行调整。具体来说，就是要修改Offset 和 Size两个属性的值，保证碰撞体的体积与平台图像精灵相符。最后进入游戏模式来测试这个平台，让玩家角色站在平台上，如图6.4所示。

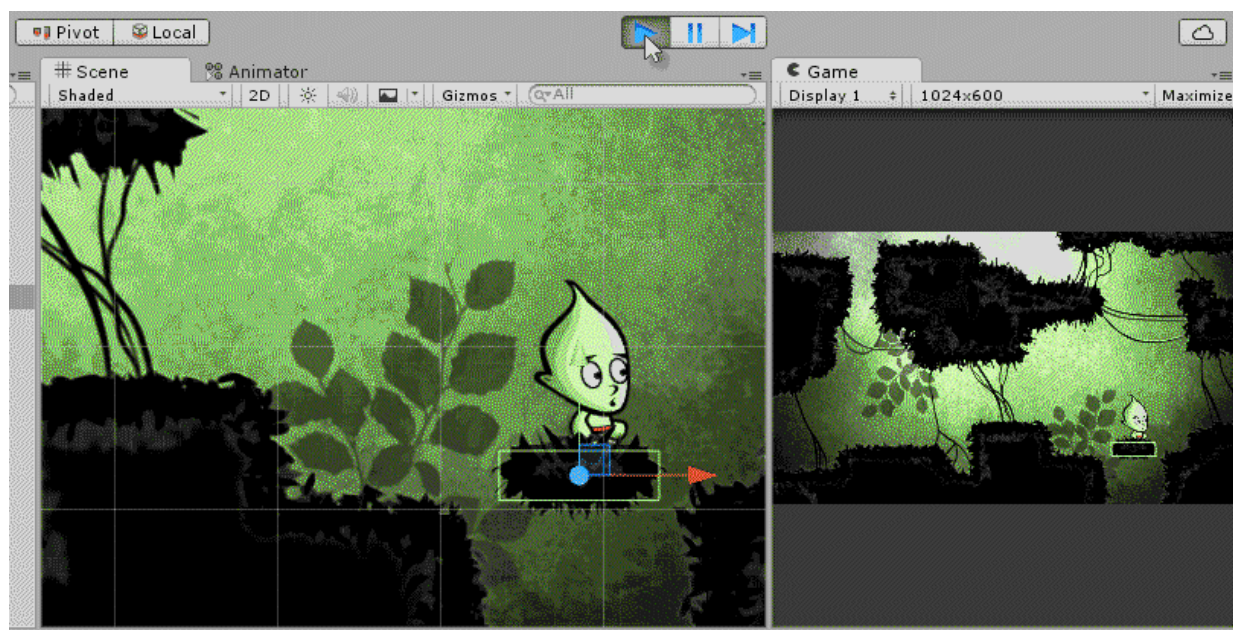


图6.4 测试平台的碰撞体（Collider）

到现在为止，这个平台都是静态的，而且不能运动，但是在设计中这个平台应该是上下不断移动的。为了解决这个问题，可以通过在菜单中依次单击“Window | Animation”，打开“Animation Editor”来创建一个预定义的动画序列。不过，在这个游戏设计中，要使用脚本文件来实现平台的动态。通常在制作动画的时候，需要在“C# Animations”或者“Baked Animations”两者之间做出选择。通常，如果一个动画并不复杂，要应用到很多对象上，并且在应用到不同的对象时可以变化。下面给出的PingPongMotion.cs脚本应该附加到这个平台上，脚本的具体内容如代码示例6.1所示。

### 代码示例6.1:

```
using UnityEngine;
using System.Collections;
//-----
public class PingPongMotion : MonoBehaviour
{
```

```

//-----
//对象的transformation属性
private Transform ThisTransform = null;

//原始位置
private Vector3 OrigPos = Vector3.zero;

//移动的轴
public Vector3 MoveAxes = Vector2.zero;

//速度
public float Distance = 3f;
//-----
// 初始化函数
void Awake ()
{
    //获取 transform组件
    ThisTransform = GetComponent<Transform>();

    //Copy original position
    OrigPos = ThisTransform.position;
}
//-----
// Update函数在每一帧被调用一次
void Update ()
{
    //使用ping pong函数来更新平台的位置
    ThisTransform.position = OrigPos + MoveAxes *
Mathf.PingPong(Time.
time, Distance);
}
//-----
}
//-----

```

下面对代码示例进行总结。

- PingPongMotion类负责实现游戏对象从初始点出发并来回移动。
- Awake()函数使用变量OrigPos来记录游戏对象的初始位置。
- Update()函数要使用Mathf.PingPong函数来实现一个值在最小值和最大值之间的光滑过渡。这个函数可以随着时间反复地在最大值和最小值范围内波动一个值，从而允许线性地移动对象。关于这

个函数的更多信息，可以访问Unity3d的在线文档

<http://docs.unity3d.com/ScriptReference/Mathf.PingPong.html>。

已经完成的代码应该附加到场景中的平台对象上，而且可以很容易地应用到其他应该上下移动（或者左右移动）的对象上。

## 6.2 创建其他的场景——关卡2和关卡3

与之前在书中创建的游戏有所不同，当前的冒险游戏将跨越多个场景。这也就是说，游戏中会包含多个不同的屏幕，玩家可以从一个屏幕的边缘进入另一个屏幕。为了引入这个功能，将要面对Unity中一些新的而且也很有趣的问题，这些问题非常值得我们去深入的研究，这也正是我们即将要做的。现在，开始为游戏制作第二个和第三个场景，这需要用到剩下的背景和前景对象，而且需要为每一个关卡配置碰撞体，允许玩家预设体在不同的环境中无缝工作。创建一个带有碰撞体的场景的详细过程已经在上一章中进行了介绍，最后，完成的场景如下。

- 关卡2在垂直方向上包含了两部分平台，下面的平台包含了一组可移动的平台。这些平台可以利用上一章中创建的移动平台预设体来实现。目前，上面的平台对于玩家来说是无害的，但是需要对其进行修改。在上面的平台上添加一些会对玩家造成伤害的炮塔对象。玩家对象可以从第一个初始关卡的最左侧进入到这个关卡，如图6.5所示。





图6.5 场景2——危险的平台和移动的平台

- 玩家可以在第一个初始关卡处，走到屏幕的最右侧进入到关卡3。这个关卡中包含了一个有房子的地面。这个房子就是游戏中NPC的家，玩家要从NPC这里来接收所收集物品的任务。在本章稍后的内容中创建这个角色，如图6.6所示。

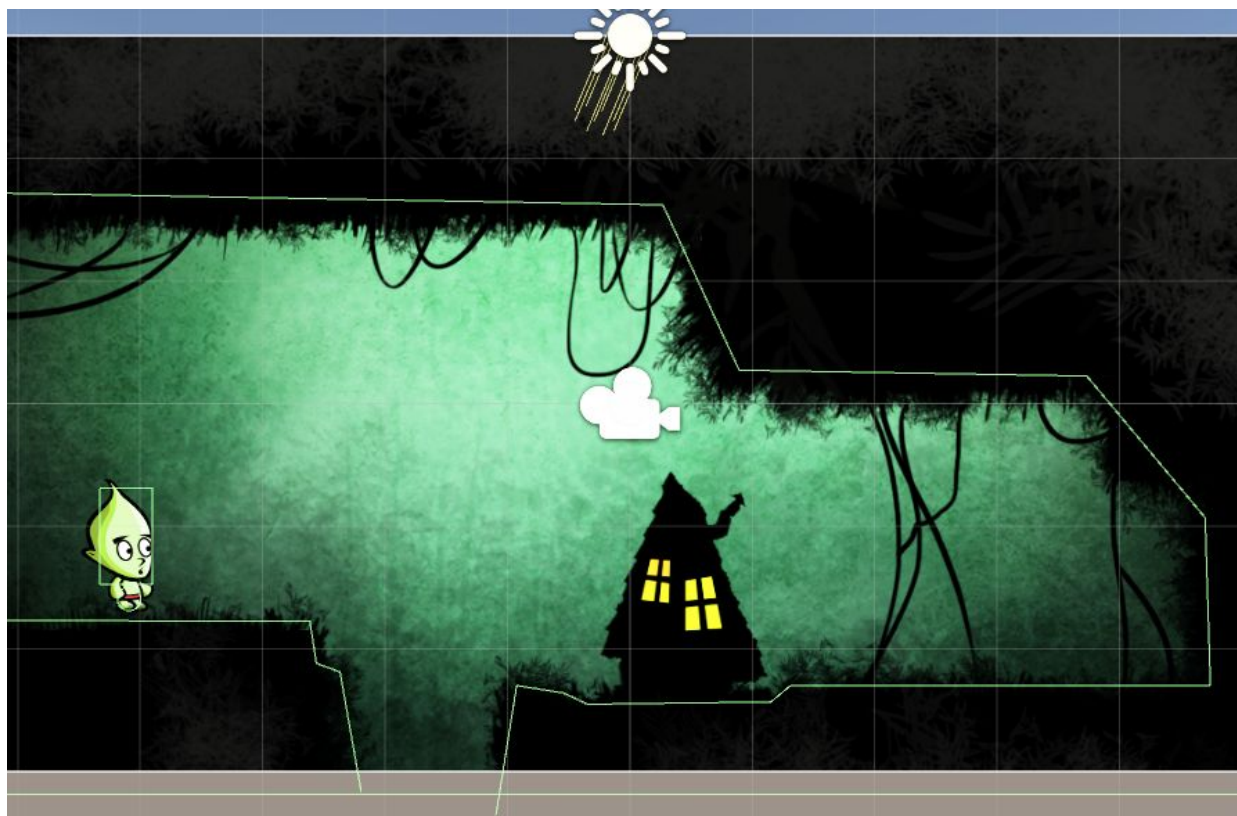


图6.6 场景3——一座孤零零的NPC住所

现在的关卡2和关卡3都是使用迄今学过的技术创建出来的。不过，为了让每个场景拥有独特的魅力和个性，必须向这些场景中添加一些特有的元素。这些元素一部分是某些场景专用的，另外一部分则是通用的。下面依次来了解这些因素。

## 6.3 死亡区域

死亡区域（**Kill Zones**）是一个所有场景中都需要使用到的通用功能，不过现在还没有实现它。也就是说，在关卡中要标记出一块二维空间区域，当玩家进入到这块区域之后，将会对玩家造成伤害，甚至会杀死玩家，尤其是在处理玩家在地面上掉进了一个洞的时候，死亡



区域就显得十分有用了。因此，在每一个关卡中都需要死亡区域的设置，因为到现在为止的每个关卡的地面都有一些坑和洞。为了实现这个功能，需要在场景（无所谓哪个场景，因为会将它创建为一个可以到处复用的预设体）中创建一个新的空游戏对象。如前面所述，可以通过单击菜单处的“GameObject | Create Empty”来创建一个新的游戏对象，然后将这个对象命名为“KillZone”，它的初始位置为游戏世界的原点（0,0,0）。最后依次单击菜单选项“Component | Physics 2D | Box Collider 2D”来创建一个二维盒子碰撞体（Box Collider 2D），并将这个碰撞体附加到“KillZone”对象上。这个盒子碰撞体就定义了死亡区域。另外要记住，在对象检查（Inspector）面板处为“Box Collider 2D”组件勾选上“Is Trigger”复选框，这样这个对象就被配置成为了一个触发器（Trigger），如图6.7所示。触发器和碰撞体是不同的，碰撞体是不允许对象出现穿越其他物体这种情况的，但是触发器确实专门检测对象是否穿越，并允许以此为依据实现自定义的行为。

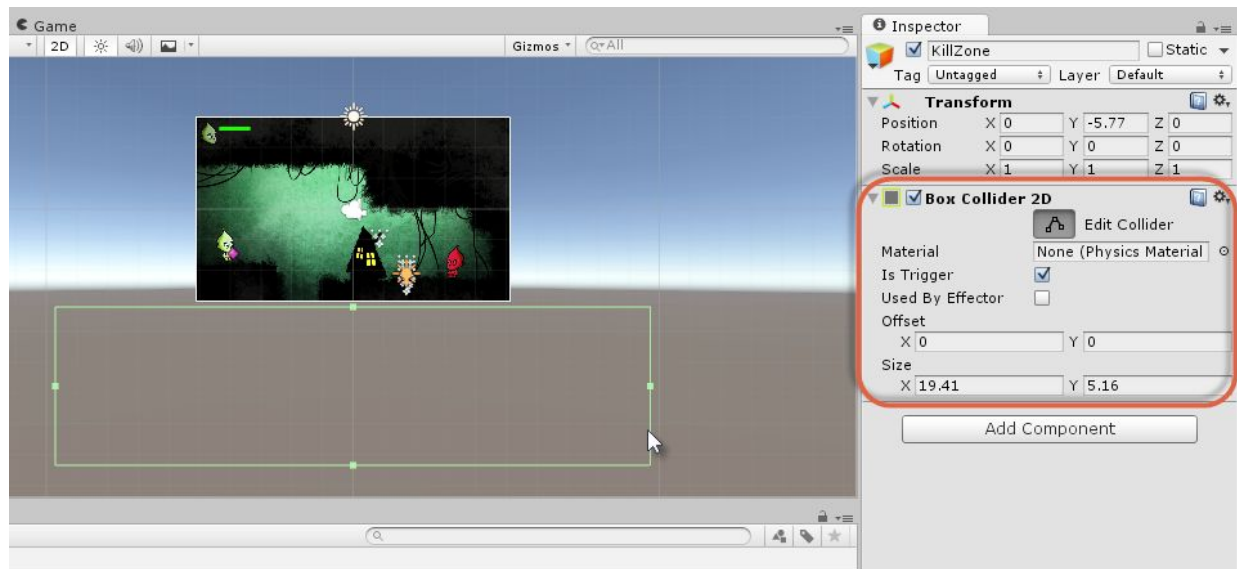


图6.7 创建一个死亡区域和触发器

接下来，将这个新的脚本命名为“KillZone.cs”，然后将这个脚本附加到场景中的“Kill Zone”对象上。这个脚本的作用就是在玩家角色一旦进入死亡区域之后，对玩家角色造成伤害。现在有多个方式来在场景中实现死亡区域。例如，一种常用的方式就是当玩家角色一旦进入死亡区域，就会立刻被破坏，另外也可以设计为当玩家角色处于死亡区域时，持续受到伤害。考虑到功能的多样性和代码的可复用性，第二种方法是最佳的选择。具体来说，就是玩家角色的生命值会按照一个特定的速度减少，直到玩家角色死亡为止，看看下面给出的代码示例6.2。

### 代码示例6.2:

```
//-----  
using UnityEngine;  
using System.Collections;  
//-----  
public class KillZone : MonoBehaviour {  
    //-----  
    //Amount to damage player per second  
    public float Damage = 100f;  
    //-----  
    void OnTriggerStay2D(Collider2D other)  
    {  
        //如果player不存在则退出  
        if(!other.CompareTag("Player"))return;  
  
        //按指定的速度对玩家造成伤害  
        if(PlayerControl.PlayerInstance!=null)  
            PlayerControl.Health -= Damage * Time.deltaTime;  
    }  
    //-----  
}  
//-----
```

下面对代码示例6.2进行总结。

- 一个Tag属性为Player的对象进入并停留在触发区域时，KillZone类负责实现持续地对这个对象造成伤害。
- 当一个具有刚体（RigidBody）组件的对象进入到触发区域之后，Unity就会自动调用OnTriggerStay2D函数，频率为每帧一次。因此，当一个物理对象进入到了死亡触发区域时，Unity会按照Update函数相同的频率调用OnTriggerStay2D函数。关于OnTriggerStay2D函数的更多详细信息，可以访问Unity的在线文档<http://docs.unity3d.com/ScriptReference/MonoBehaviour.OnTriggerStay2D.html>。
- Damage变量实现对玩家的伤害的编码，这个值会通过修改PlayerControl类中的Health静态变量来实现对玩家的伤害，当Health值为0时，玩家角色就被销毁了。
- 现在对这个游戏进行测试，在场景中标记出一块死亡区域，然后在游戏模式中让玩家走进这块死亡区域，当玩家角色走进时，就会受到伤害，甚至被销毁。为了确保玩家角色被立刻销毁，应该将伤害设置成一个非常大的值，如9000！测试完成之后，将死亡区域制作成一个预设体，方法很简单，只需要从层次（Hierarchy）面板处将“Kill Zone”拖曳到项目面板的“Prefab”文件夹中。当需要将“Kill Zone”预设体添加到各个关卡中时，对碰撞体进行修改，如图6.8所示。

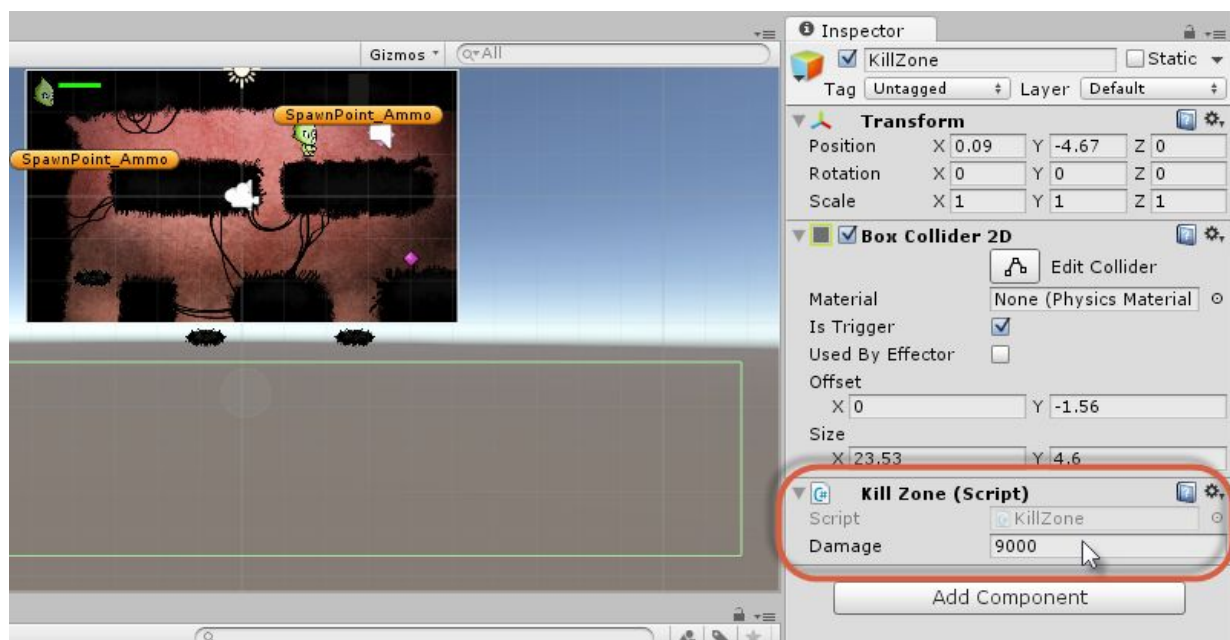


图6.8 配置一个接触以后就会毁灭的死亡区域

## 6.4 用户界面中的生命值条

上一节介绍了游戏中的第一个危险和威胁。也就是说，一个死亡区域可以破坏甚至杀死玩家角色。因此，玩家角色的生命值有可能初始状态逐渐减少。对于玩家和开发者来说，生命值的可视化是非常有用的，基于这个原因，将玩家的生命值以一个条的形式在屏幕上显示出来。这个配置好的对象也可以被制作成一个预设体，然后在更多的场景中对其进行复用，这将是一个非常有用的功能。图6.9所示为一个预览，展示工作的成果。

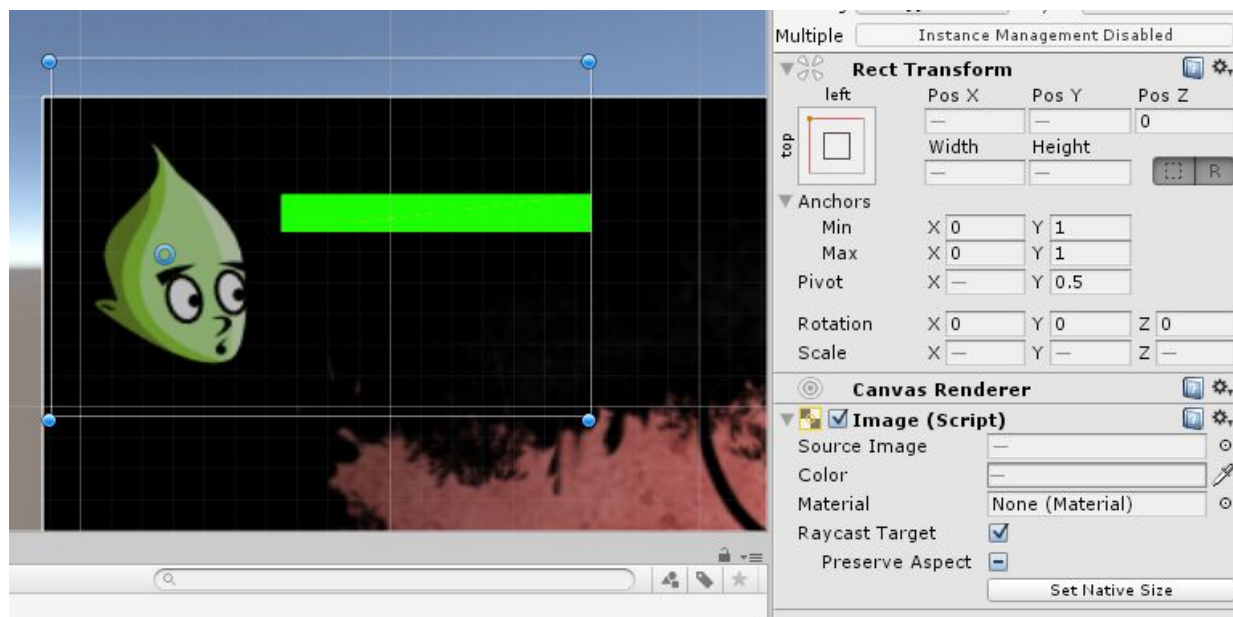


图6.9 准备创建玩家生命值

可以在应用程序菜单中依次选中“GameObject | UI | Canvas”来在场景（任意一个场景）中创建一个新的GUI画布（Canvas）。完成这个操作以后，如果当前场景中不存在Event System对象，就会自动地创建一个EventSystem对象，该对象对于UI系统的正确使用至关重要。如果不小心删掉了这个对象，可以通过在应用程序菜单依次单击“GameObject | UI | EventSystem”来重新创建。这个新创建的画布对象就是用来绘制GUI的表面的，如图6.10所示。

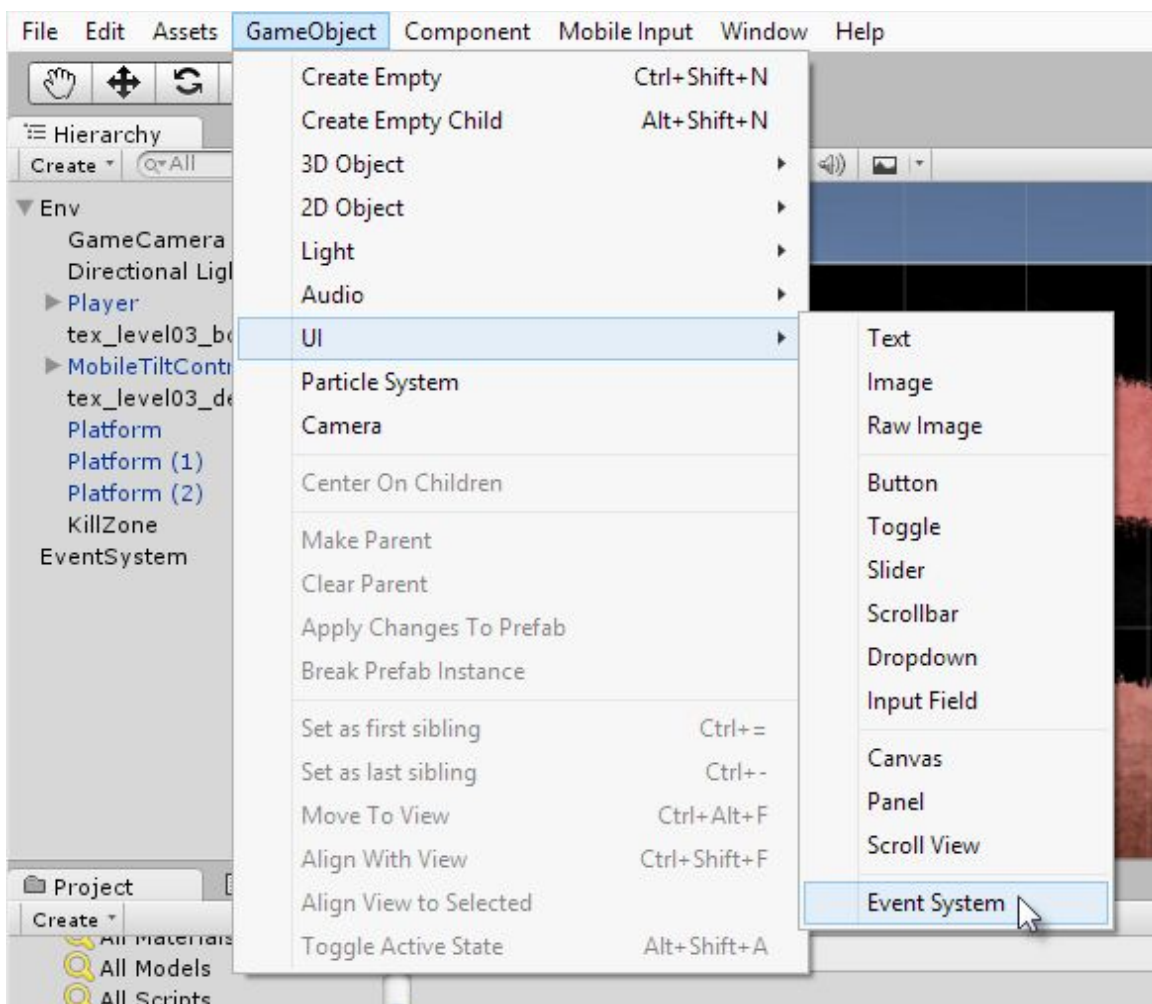


图6.10 创建一个GUI画布和一个“EventSystem”

接下来，为UI创建一个新的、专用的摄像机对象，并将其添加成为新创建画布的子对象。通过创建一个单独的UI渲染摄像头，可以将所需的摄像机的特效以及其他的图像调整选项应用到UI上。如果想要创建一个作为子对象的相机，只需要在层次结构（Hierarchy）面板上的画布对象上单击鼠标右键，然后在弹出来的上下文菜单中选中摄像机，这样就为在场景中选中的对象添加了一个作为子对象的摄像机，如图6.11所示。



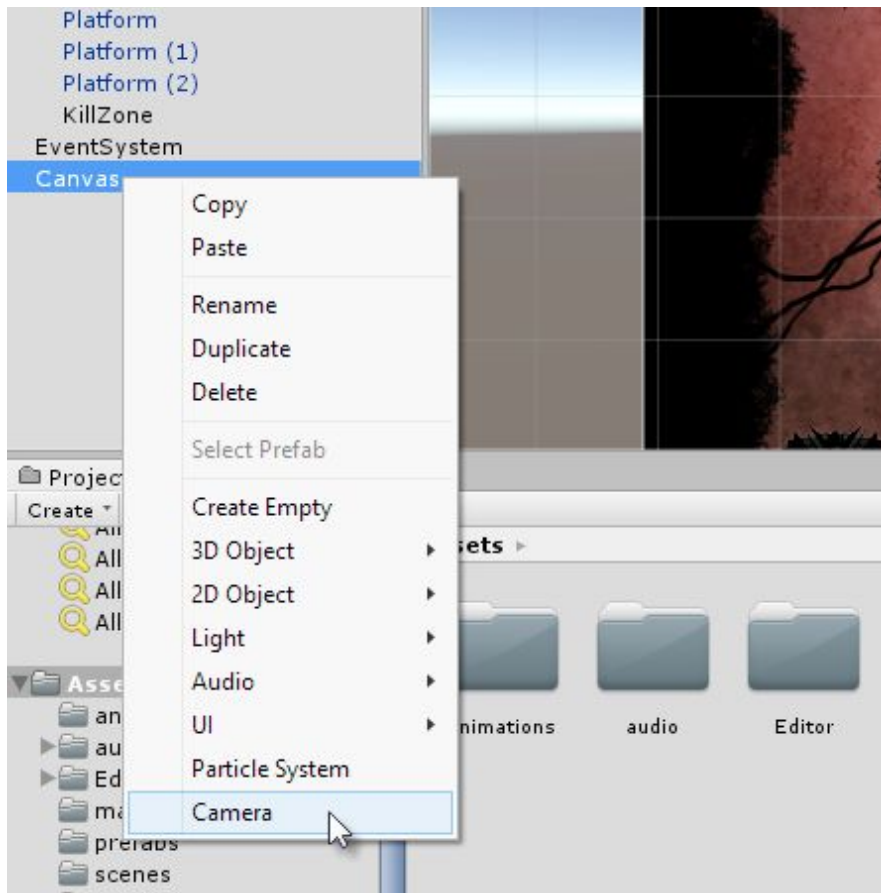


图6.11 创建一个摄像机子对象

现在将UI摄像机设置成一个正交（Orthographic）类型的摄像机，在上一章中已经学习过了设置方式。图6.12所示为如何将相机设置为正交相机。记住，一个正交相机是一个真正的二维相机，因为它消除了所有的透视和视角的渲染效果，这一点十分适合应用在GUI以及其他在屏幕中显示的对象中。此外，摄像机的Depth值可以从对象检查（Inspector）面板处找到，应该设置为一个比主游戏相机（Main Camera）更大的值，以确保它呈现在其他一切事物的顶部。否则，GUI可能会被覆盖，从而在游戏中无法正确显示。

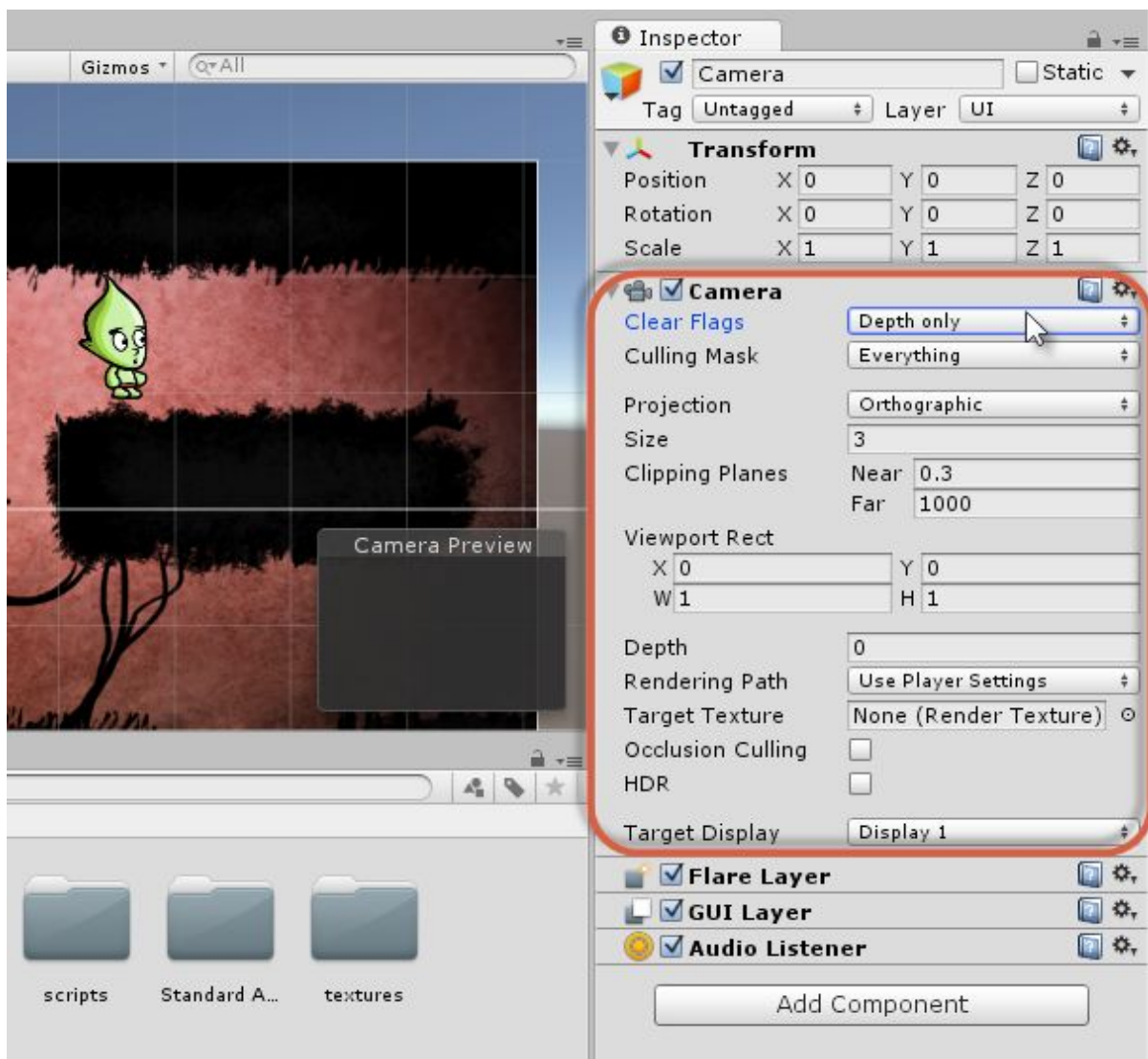


图6.12 为GUI渲染配置一个正交相机

现在创建好的摄像机已经差不多准备好了，不过，当前它和其他的摄像机一样，会对场景中的所有事物进行渲染。这也就是说，在当前的场景中所有的事物会被两个独立的摄像机渲染两次。这样不仅浪费了系统的资源，导致系统性能变差，而且其中的第二台摄像机是完全不需要的。相反，希望的是初始的摄像机渲染场景中的一切，也就是游戏中的角色和环境，但是不要渲染GUI对象。同样，新创建的GUI摄像机应该只渲染GUI对象。为了实现这个功能，首先选中游戏的主摄

像机，然后在对象检查（Inspector）面板处单击“Camera”组件处的“Culling Mask”下拉列表框，单击去掉对“UI”选项的勾选。在这个下拉列表中可以指定哪些图层不进行渲染，如图6.13所示。

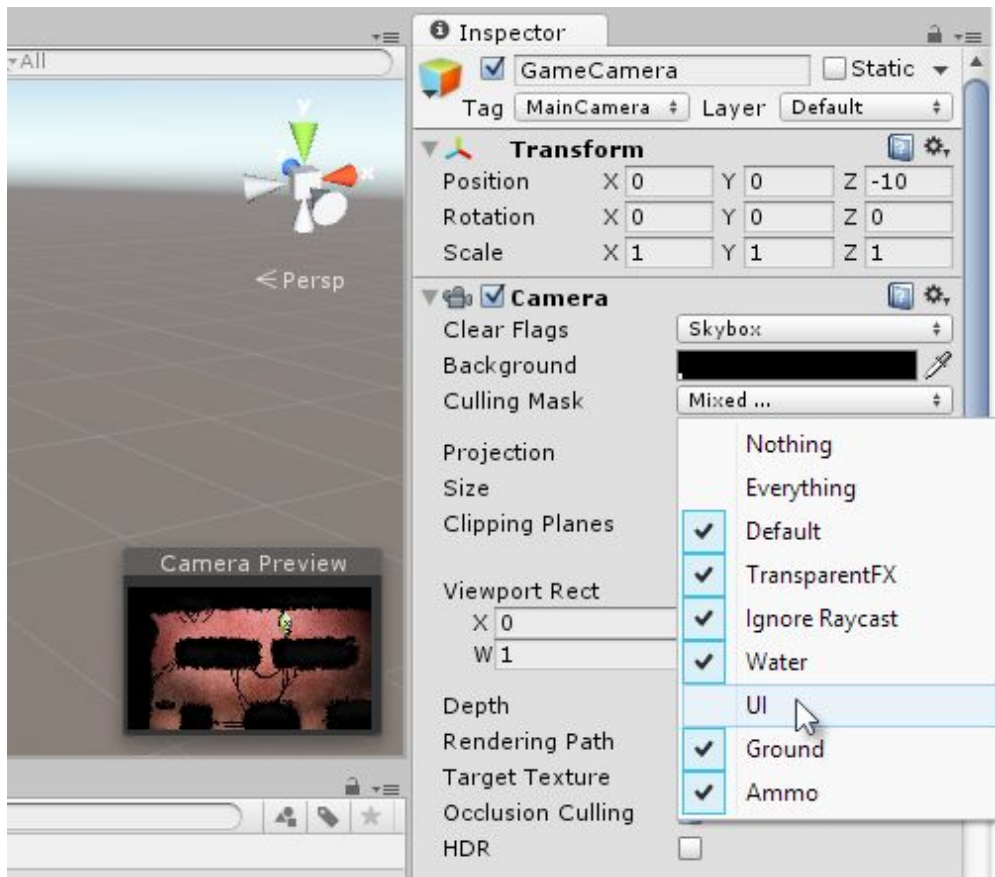


图6.13 在主摄像机指定UI层不进行渲染

选择GUI摄像机，按照同样操作在Camera组件处选择“Culling Mask”下拉列表框，首先选中“Nothing”以取消所有的选项，然后选中UI选项，这样就可以只渲染UI层，如图6.14所示。

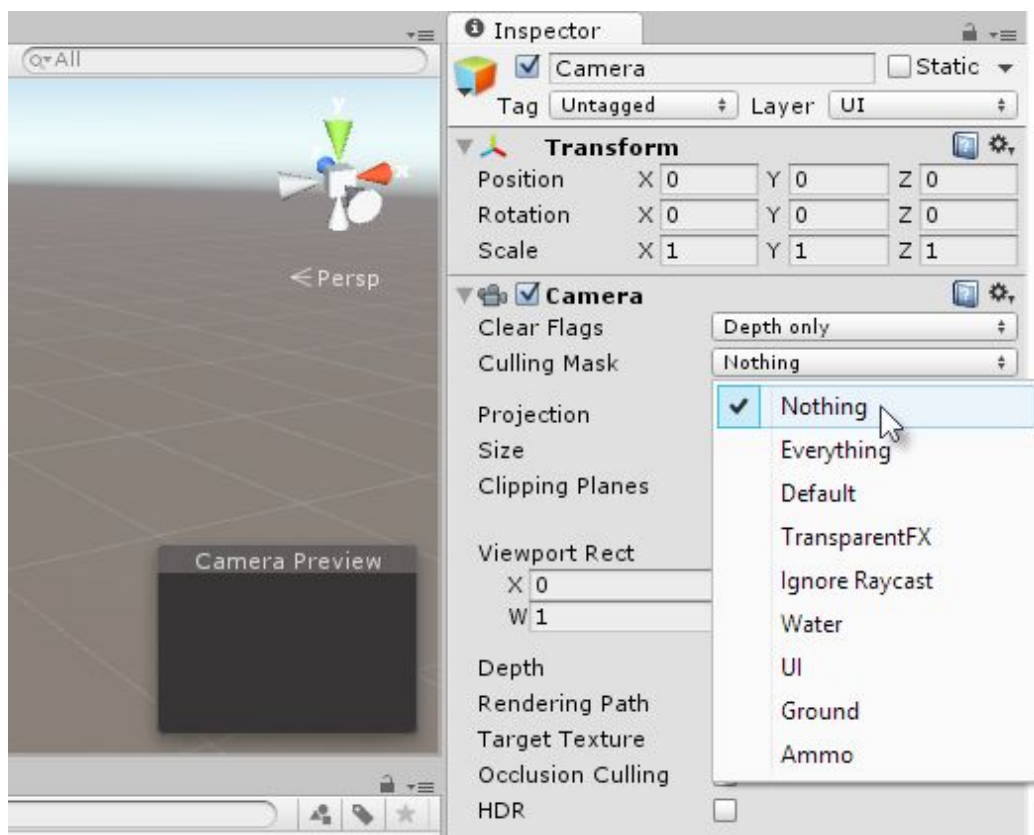


图6.14 让GUI摄像机忽略除了UI以外的所有层

默认情况下，任何新创建的画布对象都是被配置在屏幕空间的叠加模式下工作的，这就意味着它将位于场景中所有没有关联到其他相机的对象的最上层。此外，所有的GUI元素都将在这个基础上进行缩放。因此，为了让工作更简单，需要对画布对象进行配置，以配合新创建的GUI相机。首先选中画布对象，然后在对象检查（Inspector）面板中的Canvas组件处，将“Render Mode”的值由原来的“Screen Space - Overlay”更改为“Screen Space - Camera”，最后将GUI 摄像机对象拖曳到“Camera”槽位，如图6.15所示。

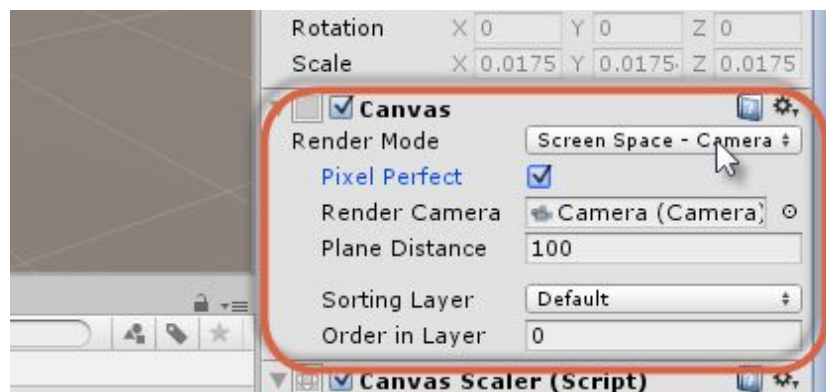


图6.15 配置用于摄像机渲染的画布组件

接下来配置Canvas Scaler组件，它是附加在画布对象上的。这个组件负责当屏幕大小发生改变时GUI的显示效果。简而言之，对于这个游戏，GUI应该跟随屏幕的尺寸变化而变化。基于这个需求，将“UI Scale Mode”下拉列表框的值设定为“Scale With Screen Size”，然后在“Reference Resolution”处设定游戏的分辨率为1024×600，如图6.16所示。

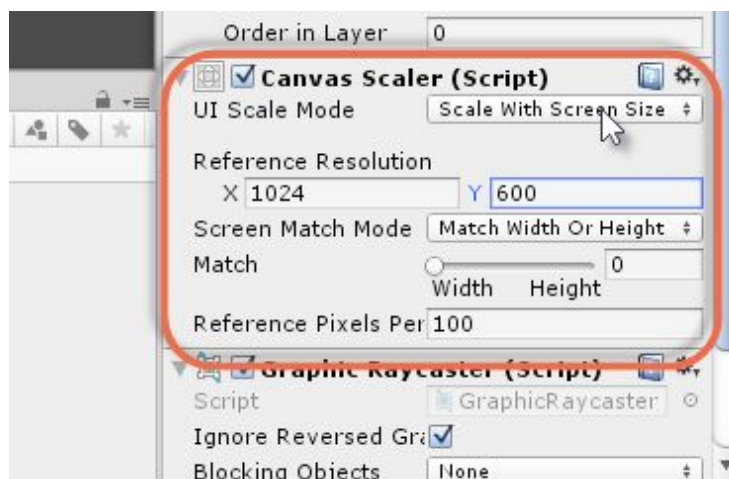


图6.16 修改画布的缩放参数以便实现响应式UI设计



将GUI组件添加到游戏中，当它们添加到场景时，就会正常显示。为了显示生命值，玩家的表示将会是十分有用的。在层次面板处的Canvas对象上单击鼠标右键，然后在弹出来的上下文菜单选中“UI | Image”，这样就可以创建一个新的“Image”对象。创建成功之后，选中这个“Image”对象，从项目面板处将玩家头部图像精灵拖动到对象检查（Inspector）面板中的“Source Image”区域处。然后，使用“Rect Transform”（键盘上的T快捷键）来改变位于屏幕左上角的图像的大小，如图6.17所示。

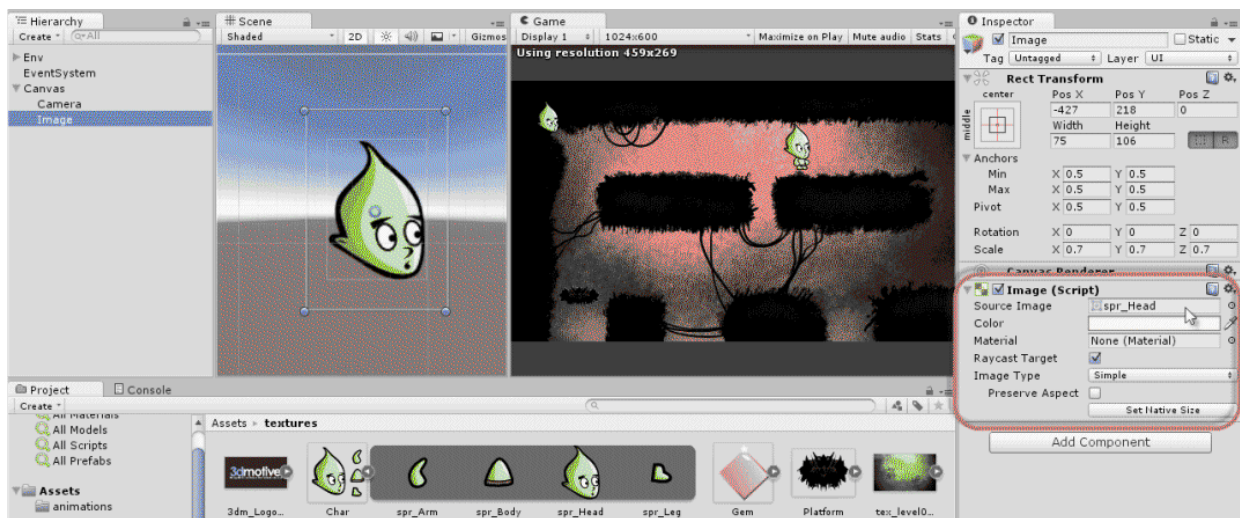


图6.17 向GUI画布上添加一个头部图像



如果没看到添加的头部图像，就将UI层分配给UI摄像机进行渲染。此外，可能需要将GUI摄像机沿着Z轴向后移动，将头部图像精灵移至摄像机的可视区域中。



最后要将头部图像精灵锚定在屏幕的左上方，在对象检查（Inspector）面板处的“Rect Transform”组件处单击“Anchor Preset”按钮，然后选中左上的对齐方式。这样就可以将头部图像精灵锁定在屏幕的左上方，确保无论哪种分辨率下，用户界面看起来都是一致的，如图6.18所示。

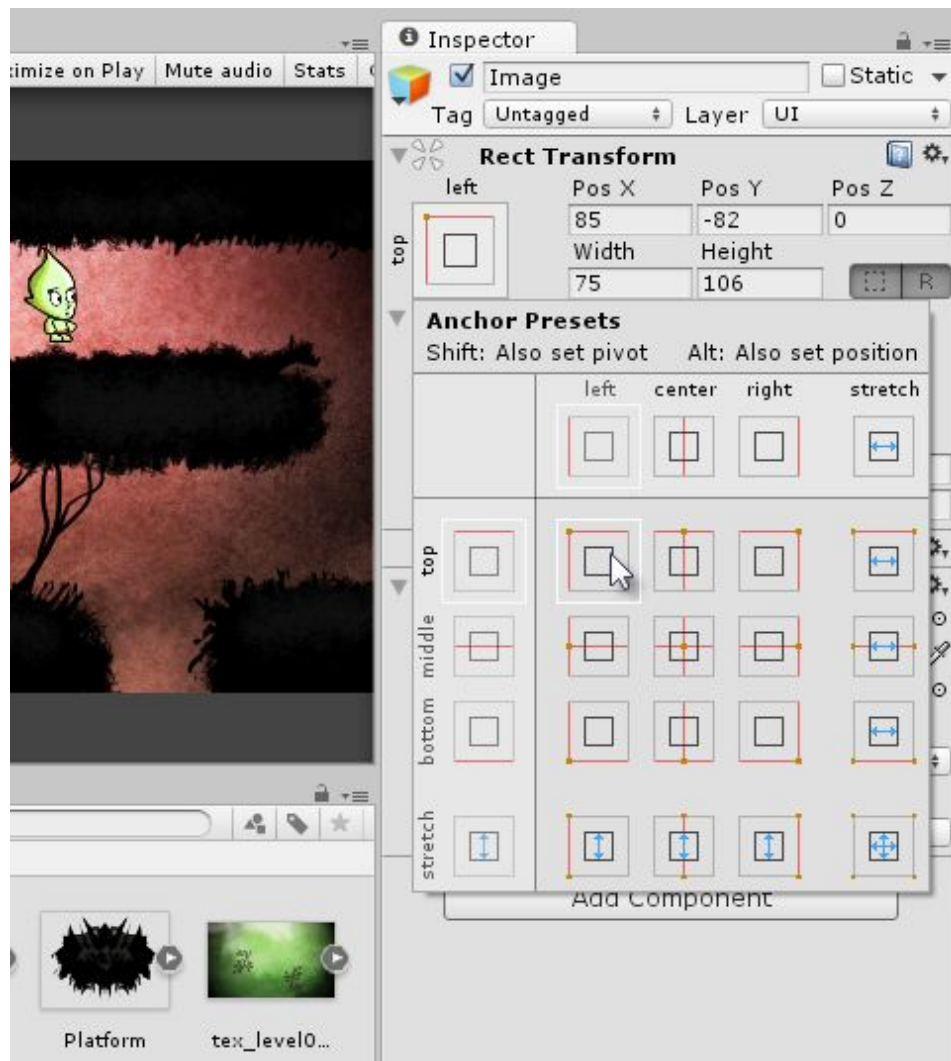


图6.18 锚定头部的位置

接下来创建一个生命条，首先向GUI画布上添加一个新的图像，添加的方法是在画布上单击鼠标右键，在弹出的上下文菜单中依次选

中“UI | Image”。接着对这个对象进行如下操作：将“Source Image”属性的值设为空，将Color属性的值设为RGB(255,0,0)。它们分别表示了背景，以及当生命值完全耗尽时的红色状态。最后，使用“Rect Transform”工具来将矩形条的大小调整为所需的尺寸，然后将它锚定到屏幕的左上方，如图6.19所示。

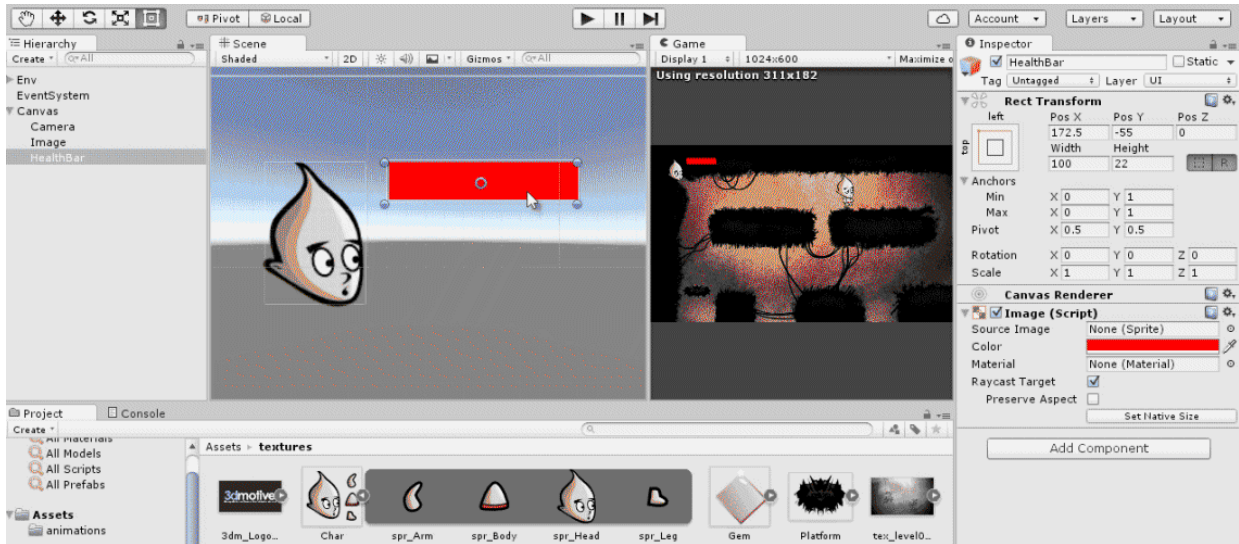


图6.19 创建一个红色的生命值状态条

为了完成状态条的制作，还需要编写脚本。具体来说，就是创建两个相同的彼此互相重叠的生命值条，其中一个为红的，一个是绿的。开始处于上面的是绿色的生命值条，当玩家角色的生命值下降的时候，它下面的红色生命值条就显示出来了。在为此功能进行编程之前，需要进行进一步的配置。具体来说，就是将生命值条的轴心由整个轴中心转移到中心靠左的位置。为了完成这个操作，需要首先选中“Health Bar”，然后在对象检查（Inspector）面板处将Pivot属性中的X的值设置为0，Y的值设置为0.5，如图6.20所示。

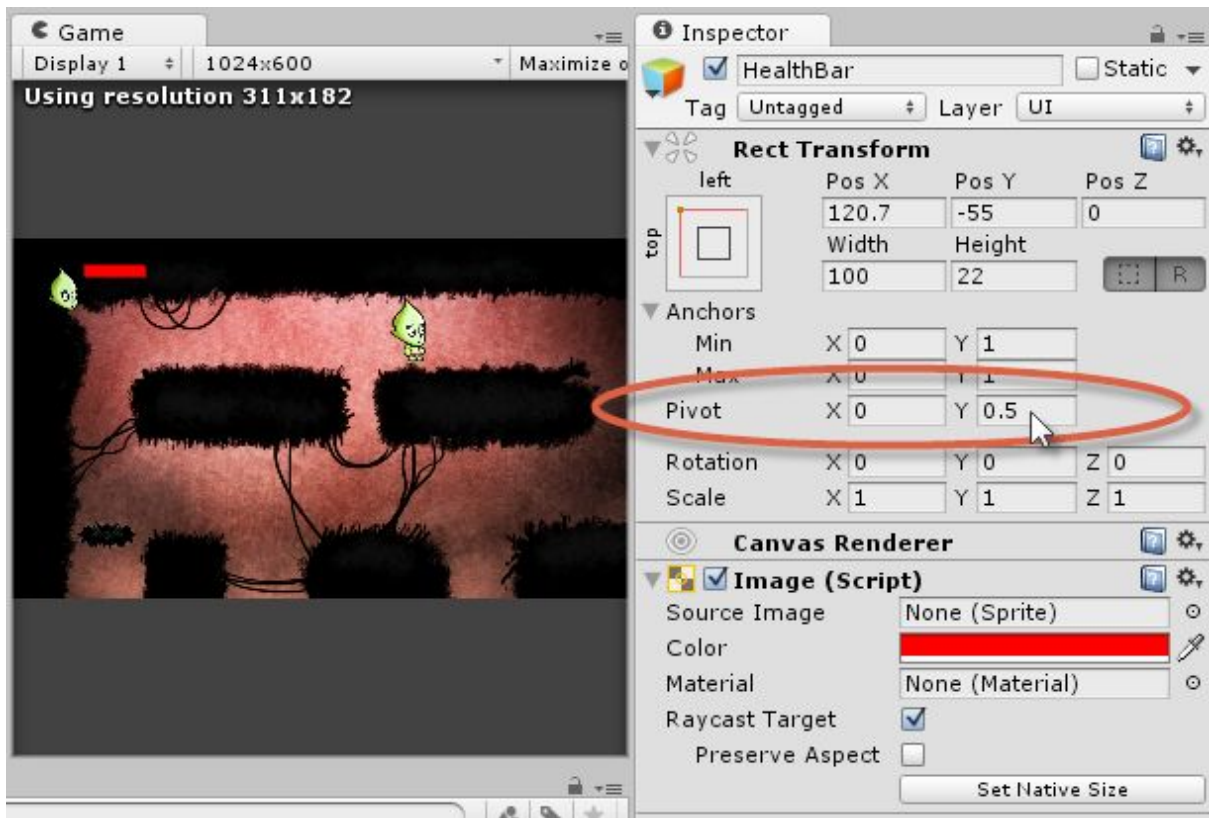


图6.20 重新定位生命值条的轴心

要创建一个代表生命值的绿色覆盖层，可以选中红色的生命值条进行复制。将复制出来的这个生命值条命名为“Health\_Green”，并将其拖曳到层次面的红色生命值条的下面，GUI元素的绘制顺序与对象的层次序号有关，序号低的对象会出现在序号高的对象的顶部，如图6.21所示。

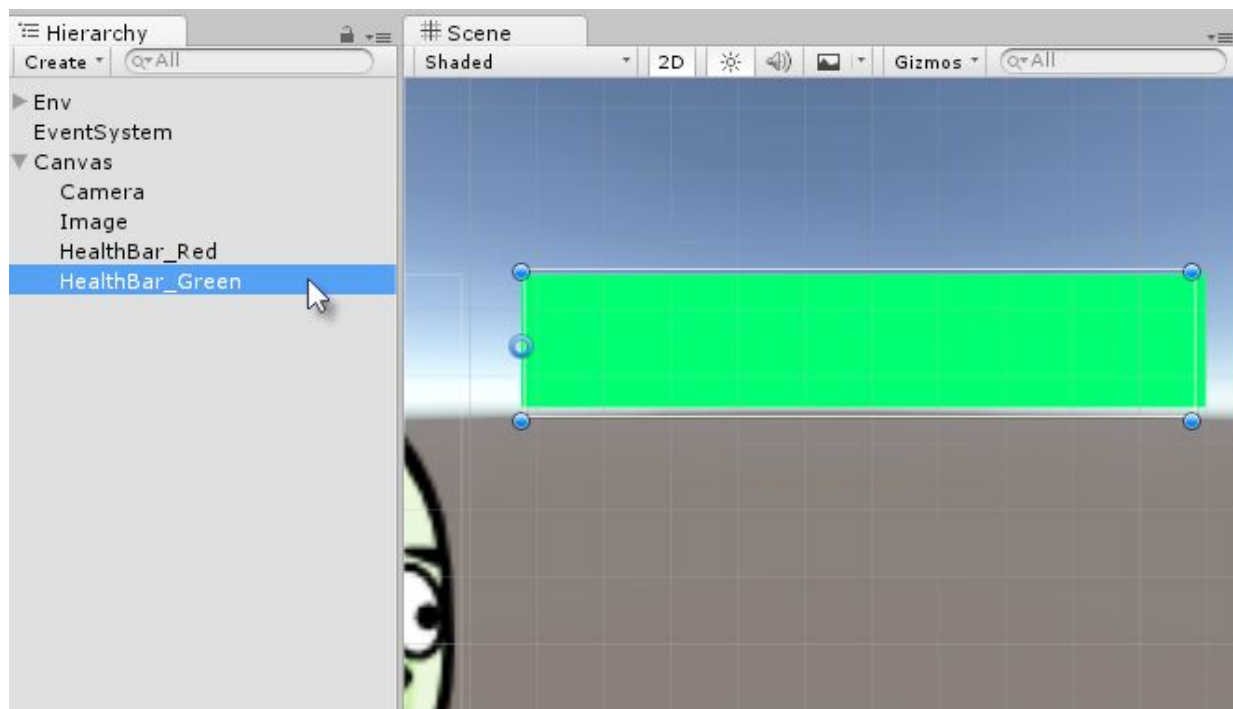


图6.21 创建一个复制的绿色条

需要创建一个将绿色生命值条的宽度与玩家角色生命值相关联的脚本文件。当玩家生命值下降时，绿色生命值条的长度就会变短，露出下面的红色条。创建一个新的脚本文件并为其命名为“HealthBar.cs”，然后将它附加到绿色生命值条上。

### 代码示例6.3:

```
using UnityEngine;
using System.Collections;

public class HealthBar : MonoBehaviour
{
    //对transform组件的引用
    private RectTransform ThisTransform = null;

    //速度
    public float MaxSpeed = 10f;

    void Awake()
    {
```

```

        //获取 transform组件
        ThisTransform = GetComponent<RectTransform>();
    }

    void Start()
    {
        //设置初始的生命值
        if(PlayerControl.PlayerInstance!=null)
            ThisTransform.sizeDelta = new
                Vector2(Mathf.Clamp(PlayerControl.
Health,0,100),ThisTransform.sizeDelta.y);
    }

    // Update函数在每一帧调用一次
    void Update ()
    {
        //更新生命值属性
        float HealthUpdate = 0f;

        if(PlayerControl.PlayerInstance!=null)
            HealthUpdate =
                Mathf.MoveTowards(ThisTransform.rect.width,
                PlayerControl.Health, MaxSpeed);

        ThisTransform.sizeDelta = new
            Vector2(Mathf.Clamp(HealthUpdate,0,100),ThisTransform.
sizeDelta.y);
    }
}

```

下面对代码示例6.3进行总结。

- **HealthBar**类负责根据玩家角色的生命值来减少绿色生命值条（位于顶部）的长度。
- **RectTransform**对象的**SizeDelta**属性决定了这个对象的长度。关于这个属性的详细信息，可以访问Unity的在线文档 <http://docs.unity3d.com/462/Documentation/ScriptReference/RectTransform-sizeDelta.html>获取。
- 函数**Mathf.MoveTowards**用来将生命值条随着时间从当前的长度平滑、渐进地转换到目标长度。也就是说，当玩家角色生命值下降

时，生命值条将会逐渐减少，而不是一下子就降低。关于这个函数的详细信息，可以访问Unity的在线文档<http://docs.unity3d.com/ScriptReference/Mathf.MoveTowards.html>获取。

最后将UI对象制作成一个预设体，从层次面板上将最顶端的Canvas对象拖曳到项目面板的Prefab文件夹中，这样就可以在多个场景中对UI系统进行复用。

## 6.5 炮弹和伤害

关卡2是一个充满危险的地方。在这个关卡中不仅仅有深坑和洞之类的死亡区域，另外还有一些例如炮塔之类的会对玩家角色造成伤害的危险物体。如何创建这些危险物体将会是这一节的主要内容。下面先来创建一个炮塔。本书的配套文件中并不包含一个炮塔的贴图或者图像，但是可以使用黑色剪影风格来创建出一个基于基本图形的炮塔道具。首先创建一个新的立方体对象（方法是依次单击“GameObject | 3DObject | Cube”），然后将它的大小调整到与炮塔的尺寸相仿，并定位到场景中靠上的悬崖上，使它成为场景的一部分，如图6.22所示。注意，也可以使用“Rect Transform”工具来调整基本图形的大小。



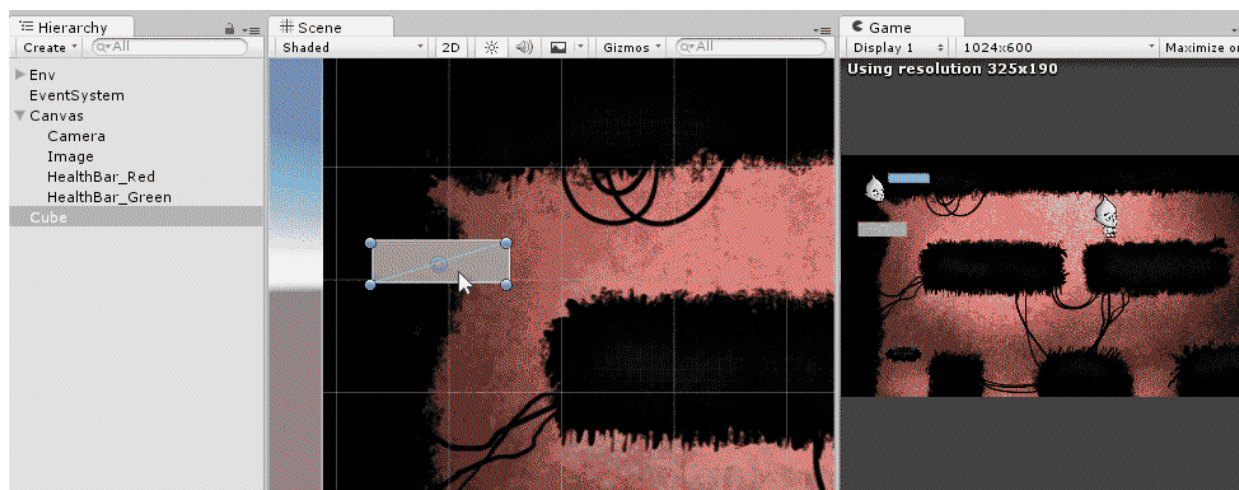


图6.22 创建一个炮塔道具

到现在为止所创建的炮塔外观还是一种明显的灰色。为了改善这个外观，需要创建一个新的黑色材质。在项目面板上单击鼠标右键，然后在弹出来的上下文菜单中依次选中“**Create | Material**”。在对象检查（**Inspector**）面板处为“**Albedo**”设置一个黑颜色，然后从项目面板处将这个材质拖曳到场景中的炮塔上。确保黑色材质“**Smoothness**”的值已经调整为 **0**，从而避免材质变得光亮。在为炮塔分配了材质之后，炮塔就能更好地融合到整个场景之中，整个关卡的配色方案也就更加合理了，如图6.23所示。

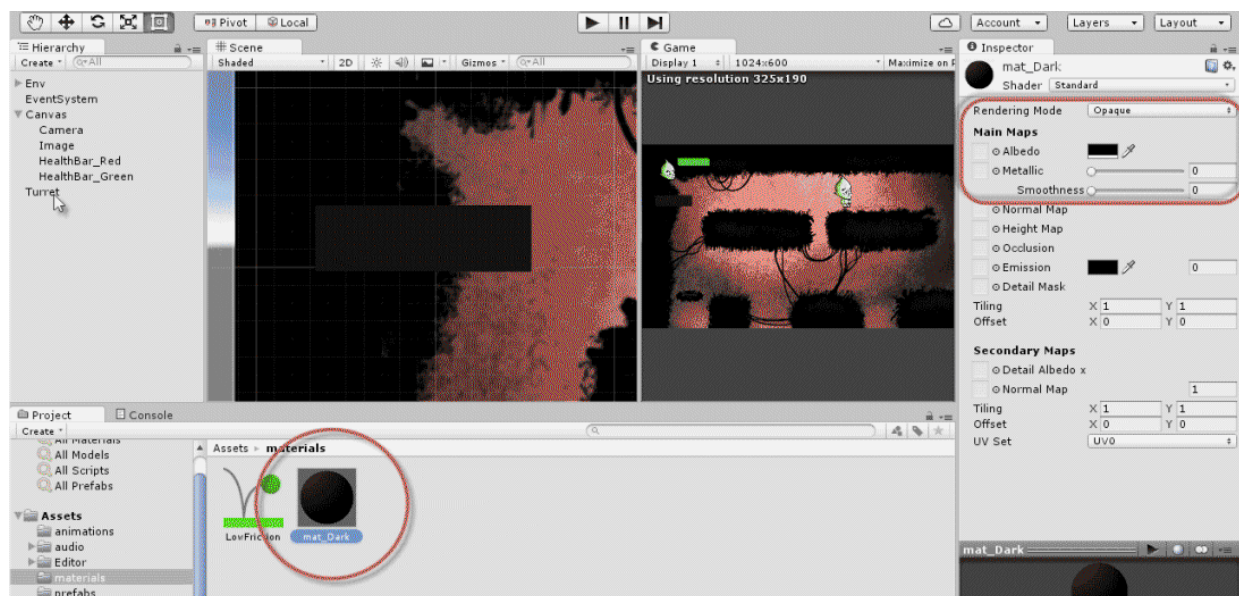


图6.23 将这个黑色的材质分配给炮塔

此时的炮塔可以发射炮弹了，还需要创建一个空的游戏对象来产生炮弹。首先依次选择“GameObject | Create Empty”，然后将层次面板中的对象拖曳到立方体炮塔上，使其成为炮塔的一个子对象。然后，将这个空对象定位到炮塔的顶部。安放到指定位置之后，为这个空对象分配一个图标，这样才能在视图中看到这个对象。确保这个空对象被选中之后，从对象检查（Inspector）面板处单击立方体图标（就在对象名字的旁边），为其分配一个图形化的表示，如图6.24所示。

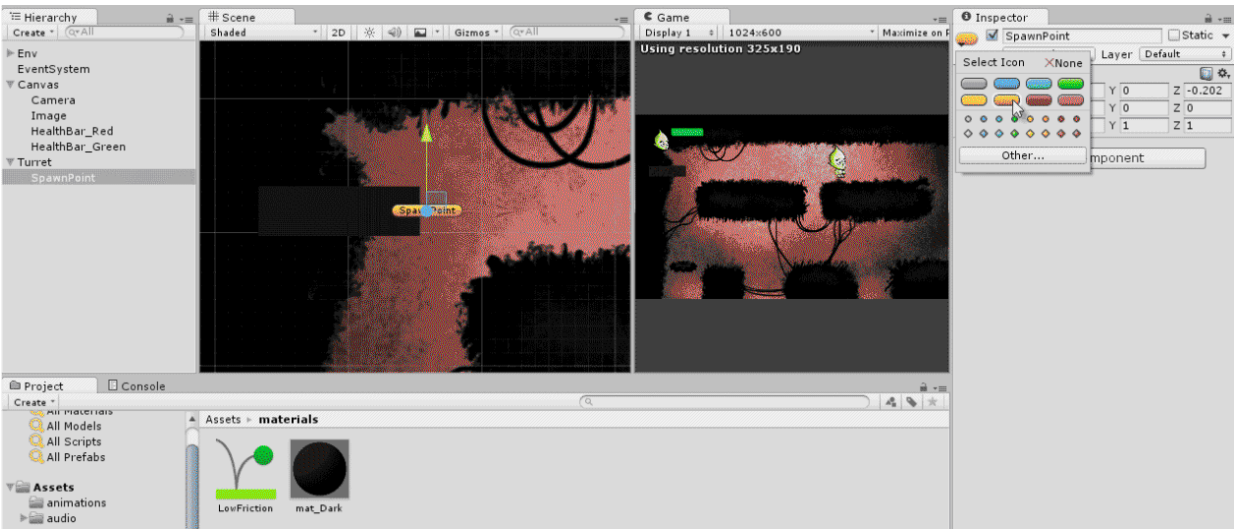


图6.24 为炮塔的发射点分配一个图标

在进一步深入研究炮弹的产生之前，需要设计一些炮弹。也就是说，这个炮塔必须发射些什么，现在就是创造这个物体的时候。炮弹的外观应该是一个发光并且闪烁的球。首先在应用程序菜单依次选择“GameObject | Particle System”来创建一个新的粒子系统。在Unity中，使用粒子系统去创建雨、火、灰尘、烟雾、闪光等特殊效果是十分有效的。当从主菜单处创建一个新的粒子系统时，就会自动在场景中创建一个对象，而且这个对象处于选中状态。当对象被选中时，就可以在Scene视图对这个粒子系统进行预览。在默认情况下，这个系统会产生一些小型斑点一样的粒子，如图6.25所示。

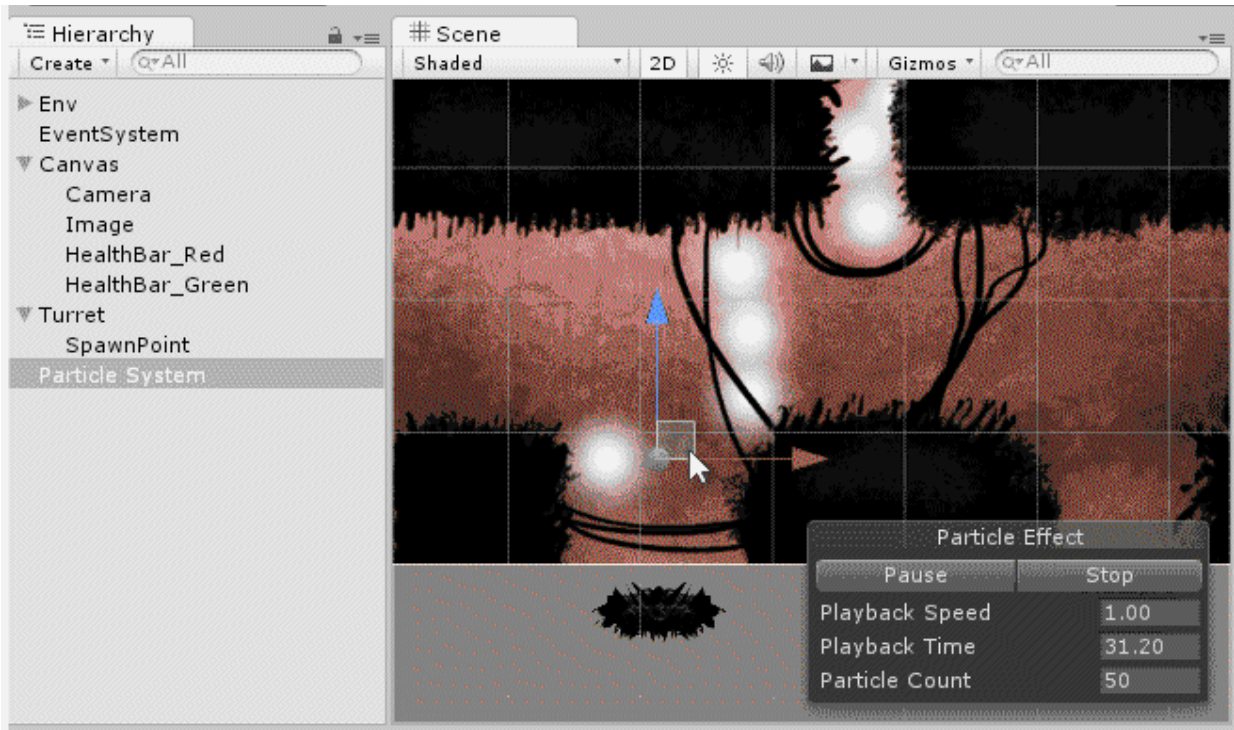


图6.25 创建一个例子系统

有时，在为二维游戏创建粒子系统之后会发现根本无法看见它，这是因为粒子系统可能被场景中的其他二维对象挡住了，例如背景或者游戏中的角色。可以在对象检查（Inspector）面板处控制例子系统的深度序号。向下滑动对象检查（Inspector）面板的滚动条，然后单击“Renderer”选项前方的展开按钮来显示更多的选项。在“Renderer”组中，将“Order In Layer”的值设置成为一个更大的值，这个值要比其他对象的值都大，这样粒子就会显示在最前面，如图6.26所示。



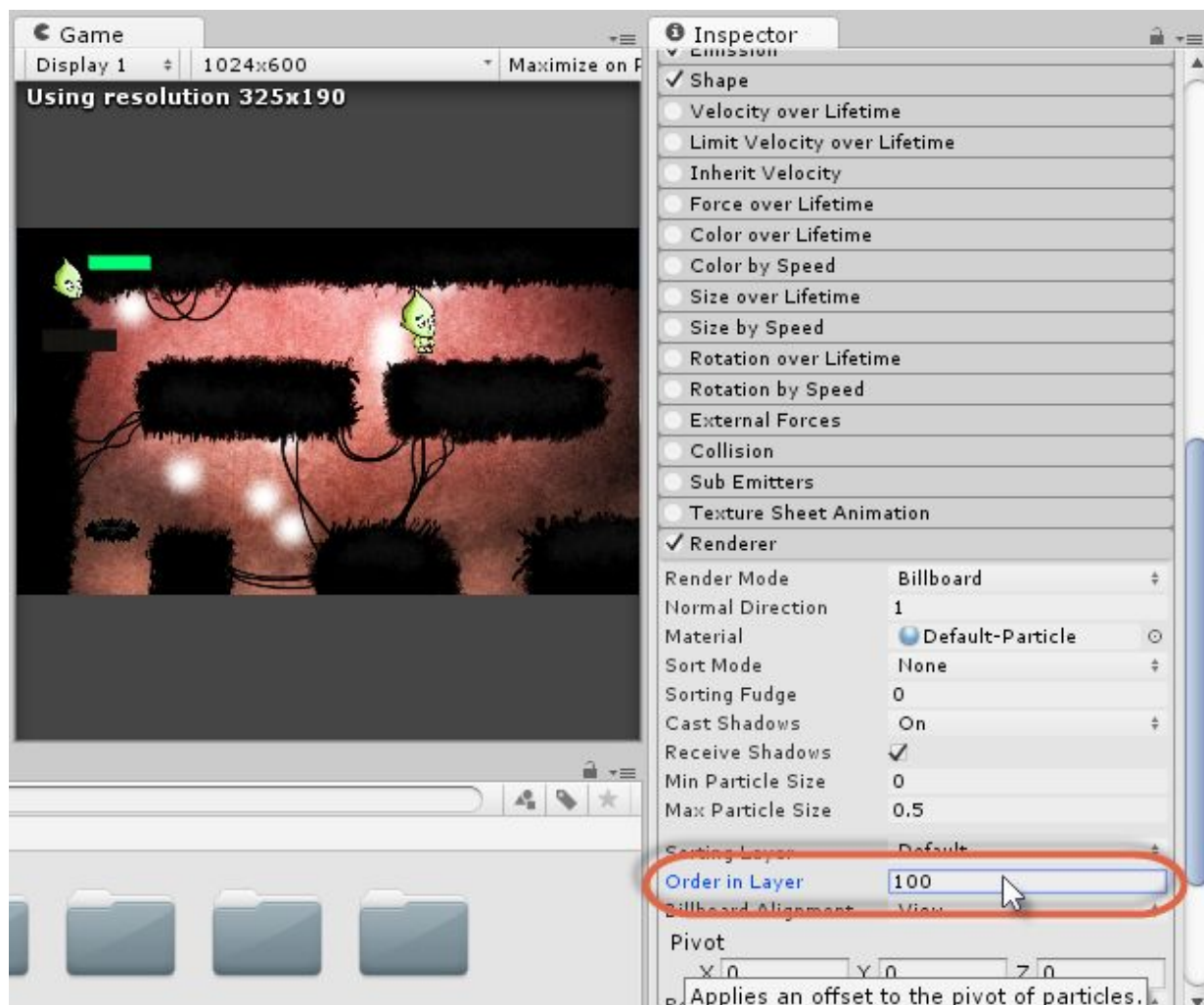


图6.26 调整粒子的渲染顺序

现在已经可以在视图中看到产生的粒子了。不过要想粒子系统如同构想的那样工作，还需要做出一些调整和改进，包括测试设置，在视图对效果进行预览，对需求进行判断，然后按需进行调整和修改。接下来创建一个更真实的炮弹对象，希望粒子是慢慢地向四面八方产生，而不是向一个方向产生。在对象检查（Inspector）面板处展开“Shape”选项来控制产生炮弹的形状。然后将“Shape”的值由“Cone”修改为“Sphere”，将Radius修改为0.01。完成之后，粒子将会产生，并从一个球面向各个方向飞行，如图6.27所示。



图6.27 修改粒子系统发射器（Particle System Emitter）的形状

对粒子系统的属性进行修改，使它出现一个能量球的效果，在对象检查（Inspector）面板中将“Start Lifetime”的值修改为0.19，将“Start Speed”的值修改为0.88，将“Start Size”的值修改为0.59，然后，将“Start Color”的值修改为“Teal”（浅蓝色），如图6.28所示。

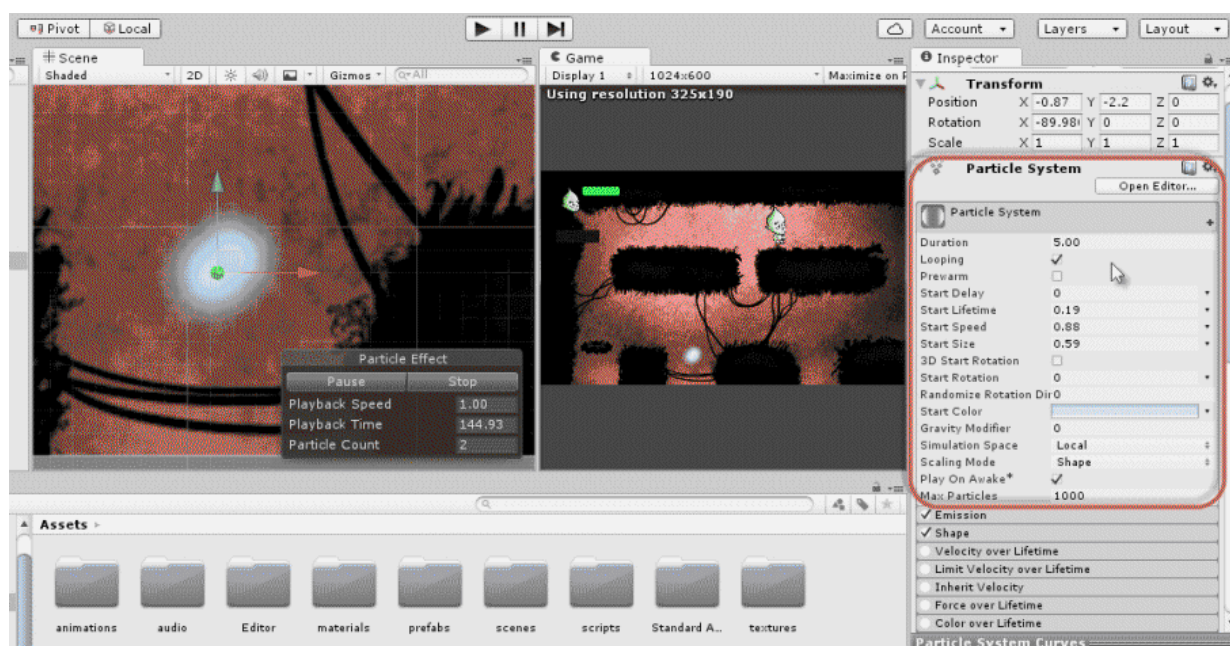


图6.28 配置粒子系统的主要属性



现在粒子系统看起来和预期的没有什么区别了。不过，如果单击工具条上的“Play”进入游戏模式以后，就会发现这些粒子根本不会动。炮弹应该在空中飞过并攻击目标，所以还需编写一个名为“Mover”的脚本，并将这个脚本附加到对象上，下面给出了Mover中的脚本内容，详见代码示例6.4。

#### 代码示例6.4:

```
using UnityEngine;
using System.Collections;
//-----
public class Mover : MonoBehaviour
{
    //-----
    public float Speed = 10f;
    private Transform ThisTransform = null;
    //-----
    // 初始化函数
    void Awake()
    {
        ThisTransform = GetComponent<Transform>();
    }
    //-----
    // Update函数在每一帧调用一次

    void Update ()
    {
        //更新对象的位置
        ThisTransform.position += ThisTransform.forward * Speed *
Time.deltaTime;
    }
    //-----
}
```

对于Mover脚本中的所有内容，我们可以说是都了如指掌了。它负责让一个对象（炮弹）按照其前进方向矢量移动。基于这个原因，再加上游戏是二维的，需要对例子系统对象进行旋转操作，保证其前向量沿着X轴的方向，如图6.29所示。

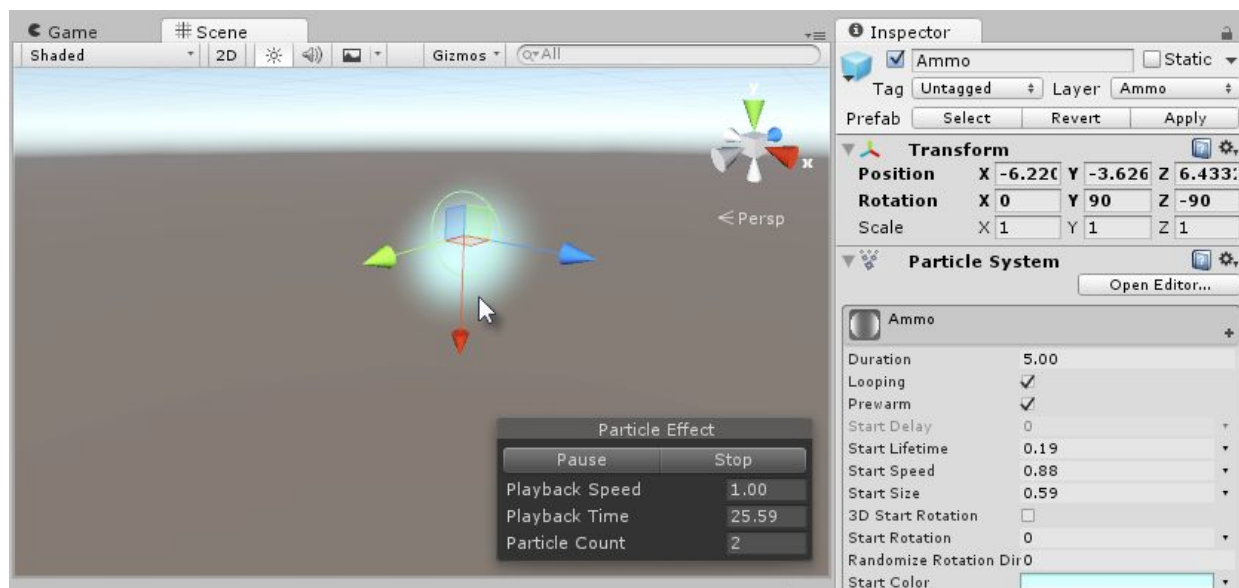


图6.29 将前向矢量对齐到X轴

接下来，这些炮弹除了在关卡中运动之外，还必须可以同玩家角色发生碰撞，并且对玩家造成伤害。要实现这个功能，还需要几个步骤，首先要向炮弹对象上添加一个刚体组件，这样炮弹就可以和其他的对象发生碰撞了。首先在场景中选中炮弹对象，然后在应用程序菜单中依次选中“Component | Physics | Rigidbody2D”，添加成功之后，在对象Inspector铭板上的“Rigidbody”组件处，勾选上“Is Kinematic”复选框。这样就确保了对象可以按照脚本移动，并且仍然不受重力的影响，可以和其他物理对象进行互动，如图6.30所示。

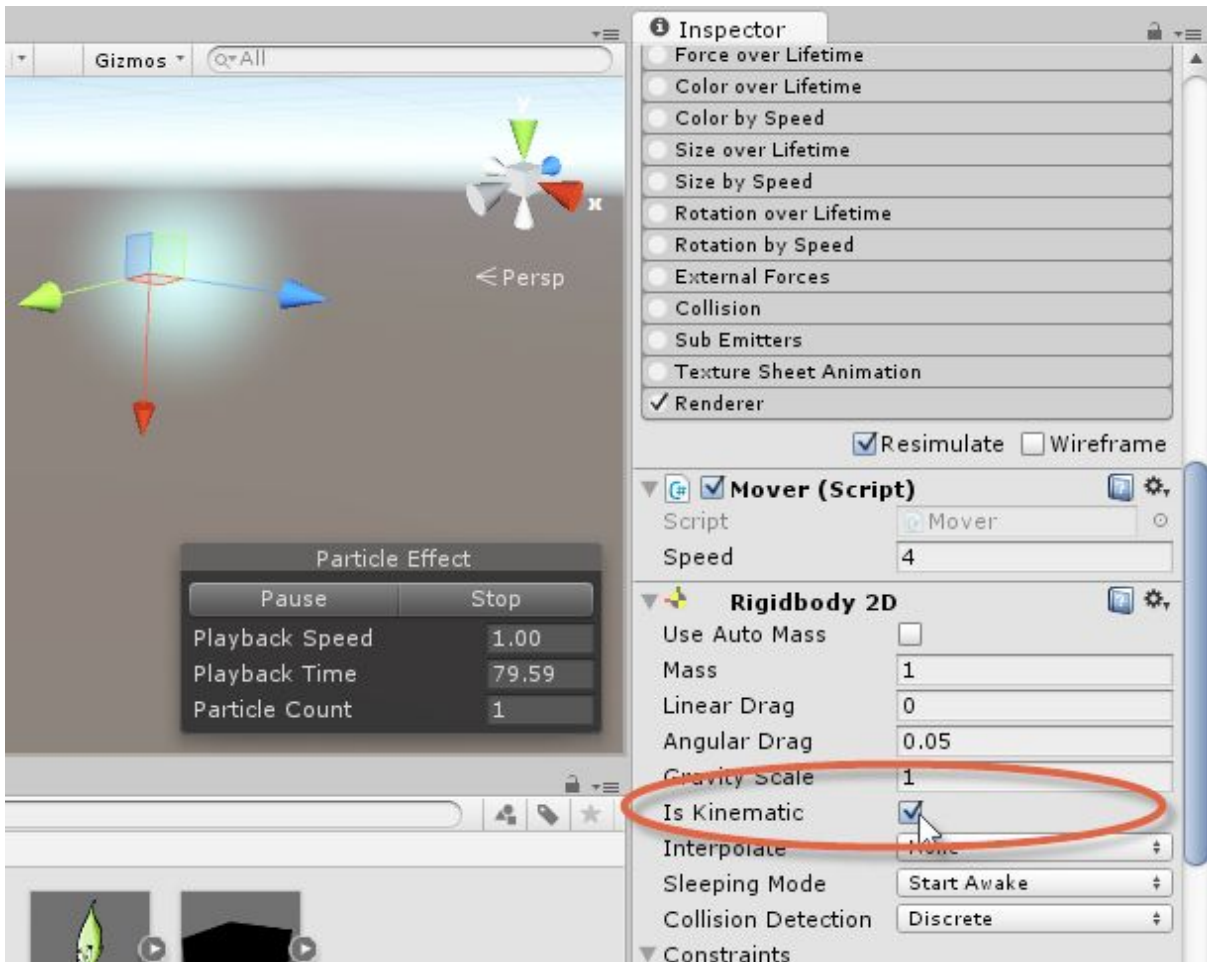


图6.30 为Rigidbody勾选上“Is Kinematic”复选框

现在为炮弹对象添加一个圆形碰撞体，这个碰撞体会决定炮弹的形状和大小等物理上的形态，添加了这个碰撞体之后，就可以检测炮弹和目标之间的碰撞。在应用程序菜单上依次选中“Component | Physics 2D CircleCollider”，成功添加之后，将这个碰撞体标记为触发器（选中“Is Trigger”），然后修改Radius值直到它接近炮弹的大小为止，如图6.31所示。

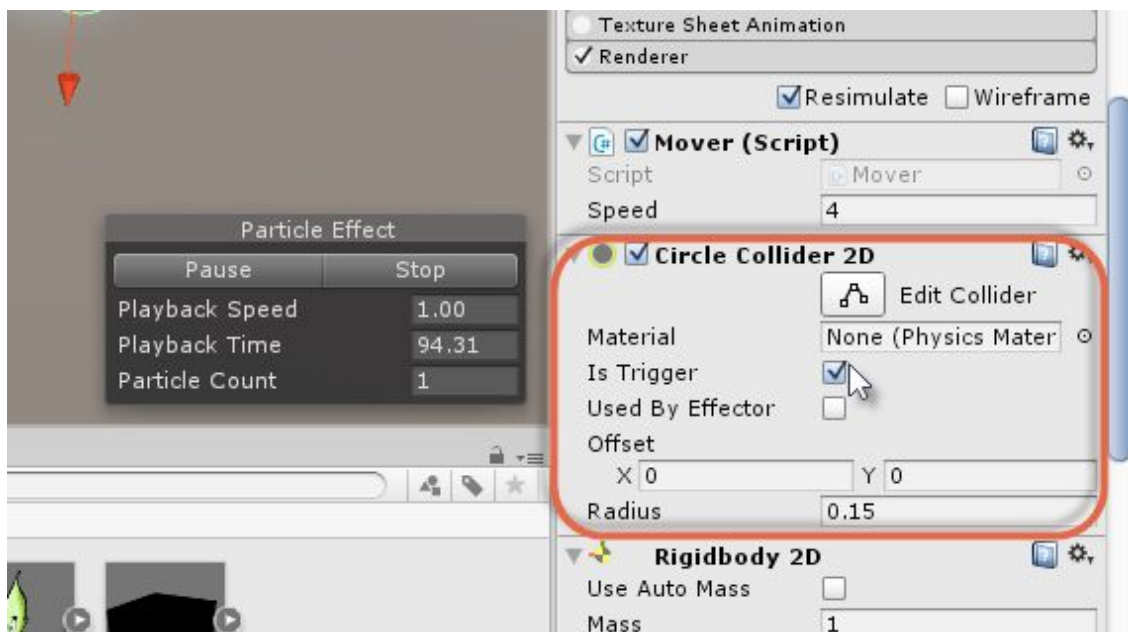


图6.31 为炮弹对象配置圆形碰撞体

现在炮弹还缺乏两个最后的功能，首先，炮弹应该可以破坏甚至毁灭它所碰到的目标；其次，炮弹应该可以自我毁灭。它们都应该发生在炮弹与目标发生碰撞之后的一段时间。为了实现这两个功能，需要编写两段新的代码，分别起名为“CollideDestroy.cs”和“Ammo.cs”。下面给出的就是Ammo.cs中的代码，详见代码示例6.5。

### 代码示例6.5:

```
using UnityEngine;
using System.Collections;
//-----
public class Ammo : MonoBehaviour
{
    //-----
    //对玩家造成的伤害
    public float Damage = 100f;

    //炮弹的生命周期
    public float LifeTime = 1f;
    //-----
    void Start()
```

```

{
    Invoke ("Die", LifeTime);
}//-----

void OnTriggerEnter2D(Collider2D other)
{
    //如果玩家对象不存在，则退出游戏
    if(!other.CompareTag("Player"))return;

    //造成伤害
    PlayerControl.Health -= Damage;
}
//-----
public void Die()
{
    Destroy(gameObject);
}
}
//-----

```

下面的代码列出了CollideDestroy.cs的内容

```

//-----
using UnityEngine;
using System.Collections;
//-----
public class CollideDestroy : MonoBehaviour
{
    //-----
    //当击中了具有相关tag属性的对象之后就会销毁
    public string TagCompare = string.Empty;
    //-----
    void OnTriggerEnter2D(Collider2D other)
    {
        if(!other.CompareTag(TagCompare))return;

        Destroy(gameObject);
    }
    //-----
}
//-----

```

这些代码所实现的功能与之前的双摇杆宇宙设计游戏基本相同。这两个文件都应该附加到场景中的炮弹对象上去。当完成以后，就从场景视图将这个炮弹对象拖曳到项目面板上的“Prefabs”文件夹中，

这样操作之后，就将这个炮弹制作成了一个预设体，可以在任何场景中再次应用，如图6.32所示。

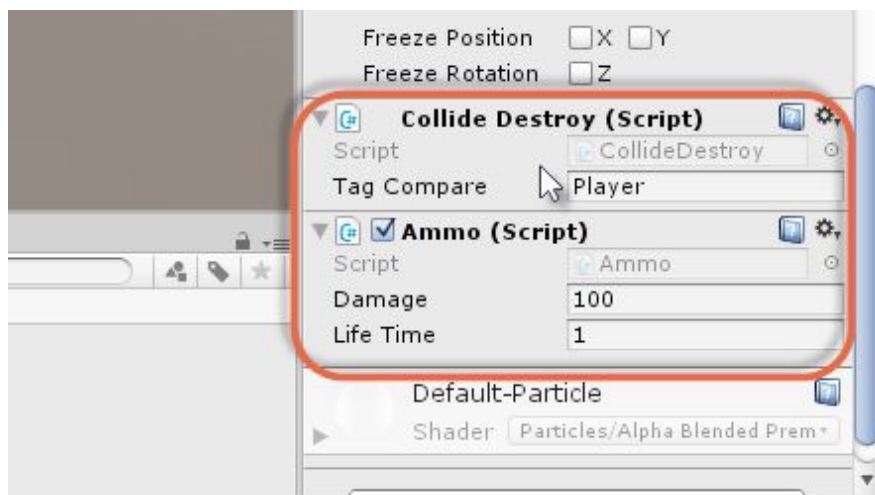


图6.32 将添加了Destory和Ammo脚本的炮弹制作成一个预设体

现在已经拥有了一个可以开火、移动以及和玩家对象碰撞的炮弹对象。将它的Damage值设置成一个足够大的值，就可以毁灭玩家对象。现在向场景中添加一个炮弹对象，然后按下键盘上的“Play”图标。当然，现在场景中还没有任何东西可以发射炮弹，接下来就开发这个功能。

## 6.6 炮塔和炮弹

现在已经创建了一个炮弹对象（发射物），而且完成了炮塔对象的设计，但是它还不能产生炮弹。下面就来实现这个功能，已经在炮塔的前方设置了一个炮弹的产生点，并且把它作为了炮塔的子对象。现在为这个对象添加一段新的脚本，将这个脚本命名



为“AmmoSpawner.cs”，这个脚本用来负责在固定的时间里产生炮弹，详见代码示例6.6。

### 代码示例6.6:

```
//-----
using UnityEngine;
using System.Collections;
//-----
public class AmmoSpawner : MonoBehaviour
{
    //-----
    //对炮弹预设体的引用
    public GameObject AmmoPrefab = null;

    //对transform的引用
    private Transform ThisTransform = null;

    //时间范围向量
    public Vector2 TimeDelayRange = Vector2.zero;

    //炮弹的生命周期
    public float AmmoLifeTime = 2f;
    //炮弹的速度
    public float AmmoSpeed = 4f;

    //炮弹的伤害值
    public float AmmoDamage = 100f;
    //-----
    void Awake()
    {
        ThisTransform = GetComponent<Transform>();
    }
    //-----
    void Start()
    {
        FireAmmo();
    }
    //-----
    public void FireAmmo()
    {
        GameObject Obj = Instantiate(AmmoPrefab,
            ThisTransform.position, ThisTransform.rotation) as
            GameObject;
        Ammo AmmoComp = Obj.GetComponent<Ammo>();
        Mover MoveComp = Obj.GetComponent<Mover>();
        AmmoComp.LifeTime = AmmoLifeTime;
```

```
        AmmoComp.Damage = AmmoDamage;  
        MoveComp.Speed = AmmoSpeed;  
  
        //等待直到下一个周期开始  
        Invoke("FireAmmo", Random.Range(TimeDelayRange.x,  
            TimeDelayRange.y));  
    }  
    //-----  
}  
//-----
```

前面的代码主要实现了使用**Invoke**函数在场景中对炮弹预设体的实例化，这段代码会在**Random.Range**产生一个随机的时间，每经过这样的一个随机时间就会调用**Invoke**函数一次。可以使用上一章炮弹实例中讲解过的对象池（**Object Pooling**）技术实现对这段代码的改进，但是在当前的这个例子中，这段代码即使不做修改也是可以接受的，如图6.33所示。

现在已经创建了一个炮塔，就像之前的炮弹一样，应该将炮塔也转换成一个预设体。首先要确认“**Time Delay Range**”（也就是炮弹产生的时间间隔）中的**X**、**Y**值设置为比0大的值。否则，炮弹就会一直不断地产生出来，从而导致玩家无法躲避。如果需要，就继续布置更多的炮塔来平衡游戏的难度。

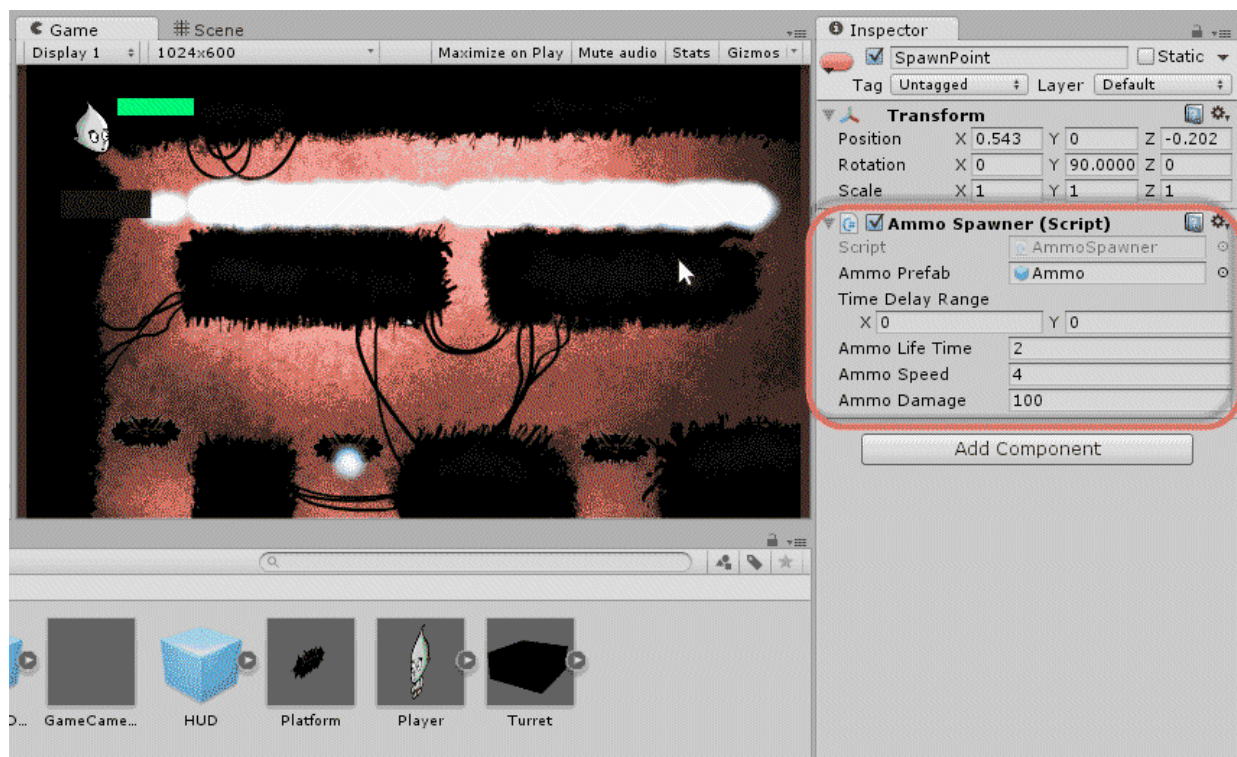


图6.33 设置“Time Delay Range”的X、Y值

## 6.7 NPC和任务系统

NPC是非玩家角色（Non-player Character）的缩写，通常指的是那些友好的、中立的非玩家控制角色。在冒险游戏中，关卡3应该有一个NPC，这个NPC就站在他的房子外面，他会提供一个任务。具体来说，在关卡2收集一个宝石，要面对很多危险，例如坑和炮塔。为了创建NPC角色，可以对玩家角色进行复制，然后修改这个角色的颜色，这样NPC角色和玩家角色看起来就不太一样了。现在只需要从项目面板拖曳“Player”预设体到关卡2的场景中，将其放置在房屋区域附近。然后将所有后添加的组件（例如玩家控制器和碰撞体）都去除，这个角色现在就变回了一个标准的不受玩家控制的图像精灵，如图6.34所示。

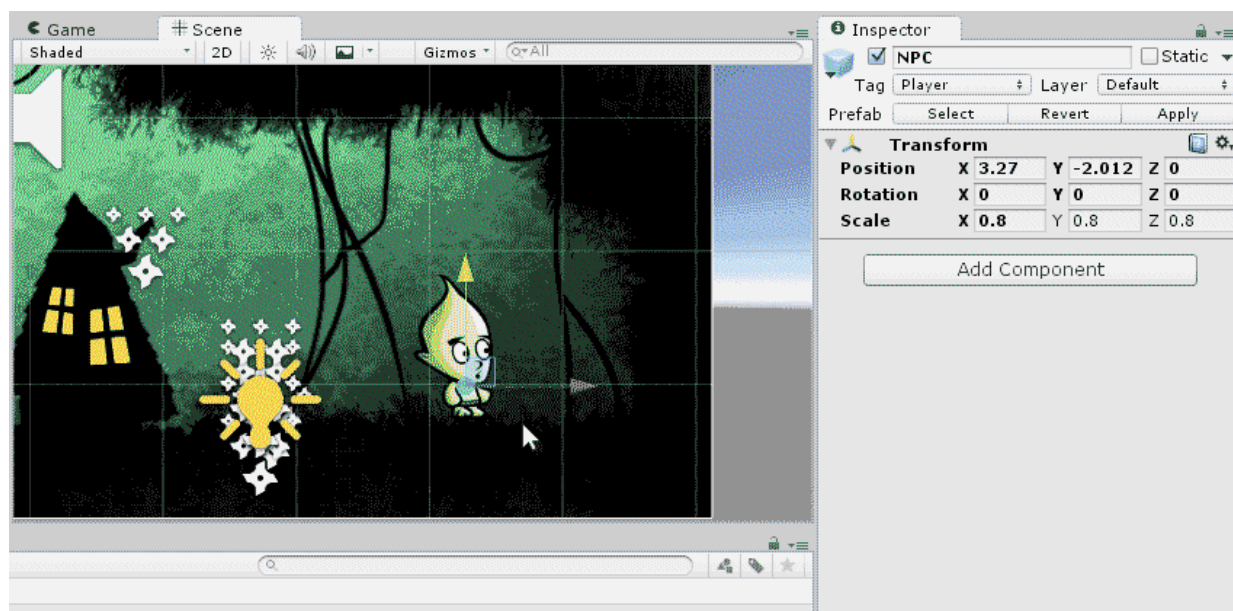


图6.34 从玩家角色预设体创建一个NPC

现在将这个角色的“Scale”属性中的X值修改为原值（0.8）的相反数（-0.8），这样这个角色的脸将会面向左面，而不是右面。注意，这里要选择整个NPC对象，而不是这个NPC的组成部分，例如手和手臂。否则所有的子对象都会翻转方向，如图6.35所示。

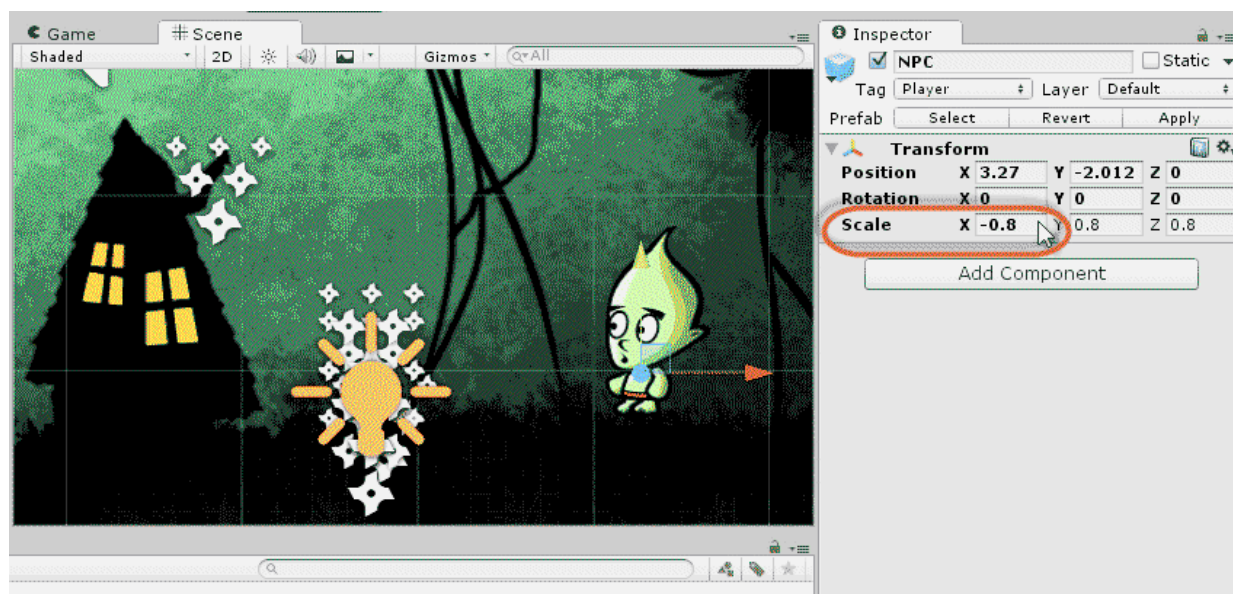


图6.35 将NPC角色的Scale属性的X值取相反

还需要改变NPC的颜色，把原来的绿色修改为红色，这样NPC看起来就和玩家角色不一样了。现在，这个角色是由几个图像精灵对象共同组成的。分别选中每个对象，然后在对象检查（Inspector）面板处修改颜色。不过如果一次性选中所有对象一起修改它们的颜色，会更加简单。从Unity 5之后，就可以一次对多个对象的共同属性进行编辑，如图6.36所示。

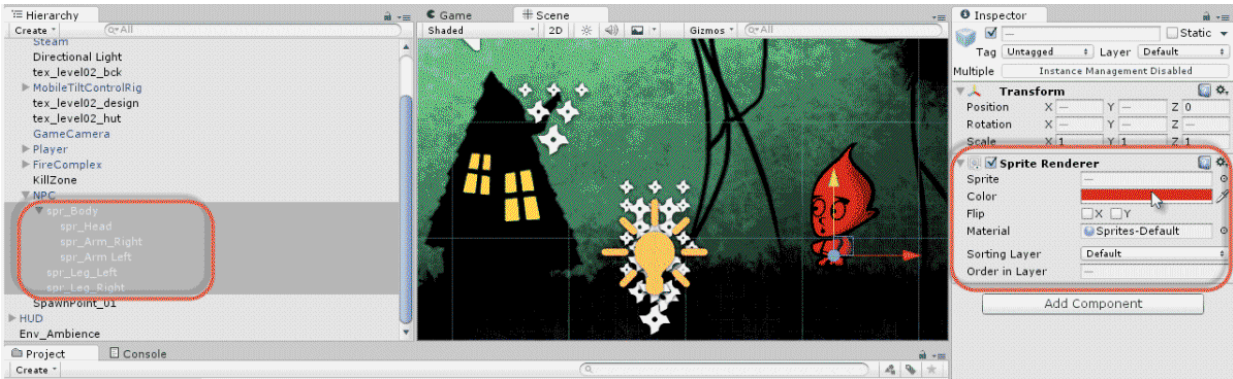


图6.36 设置NPC颜色

NPC应该能通过某种途径和玩家进行交流。这表示当玩家角色靠近一个NPC时，这个NPC应该展示一个文本对话框。这个对话框中的内容应该是变化的，这种变化是根据玩家完成任务的状态决定的。当玩家第一次访问NPC时，NPC应该给予玩家一个任务。当玩家第二次访问NPC时，NPC的回应应该与第一次不同，但是回答的内容会根据玩家当前是否完成了任务而不同。要实现功能：当玩家接近NPC时，必须可以检测到，可以使用碰撞体来实现。首先选中场景中的NPC，然后在应用程序菜单处选中“Component | Physics 2D | Box Collider 2D”，注意，这一次的碰撞体不应该与NPC大小一样，而是一个环绕在



NPC外部的区域，当玩家进入到这个区域时，就会与NPC进行交谈。将这个碰撞器设置为一个触发器（Trigger）对象，这样玩家才能进入，并且可以穿过，如图6.37所示。

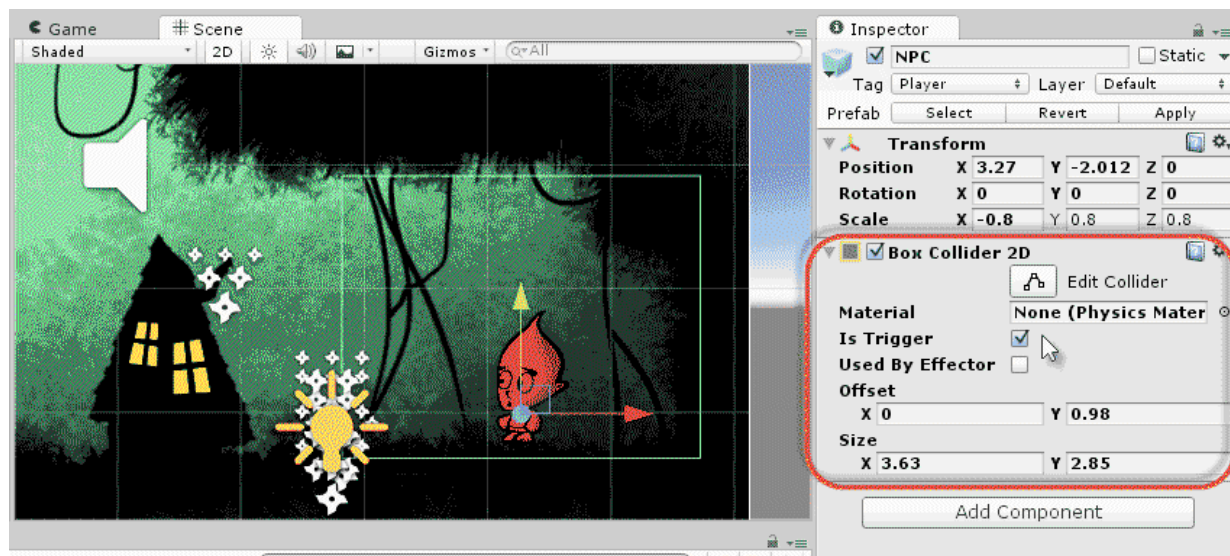


图6.37 配置一个NPC的碰撞体

在这个阶段，需要一个GUI元件作为展示NPC讲话内容的载体。这个功能只需要一个包含了文本作为子对象的GUI画布对象。通过依次在应用程序菜单处单击“GameObject | UI | Canvas and GameObject | UI | Text”来添加这些对象。这里的画布对象需要一个画布组（Canvas Group）组件，可以使用“Component | Layout | Canvas Group”选项来添加，这样就可以将所有的子对象作为一个整体来设置Alpha属性。从对象检查（Inspector）面板中修改Alpha的值，当值为1时，表示这个对象完全可见；值为0时，表示完全透明，如图6.38所示。



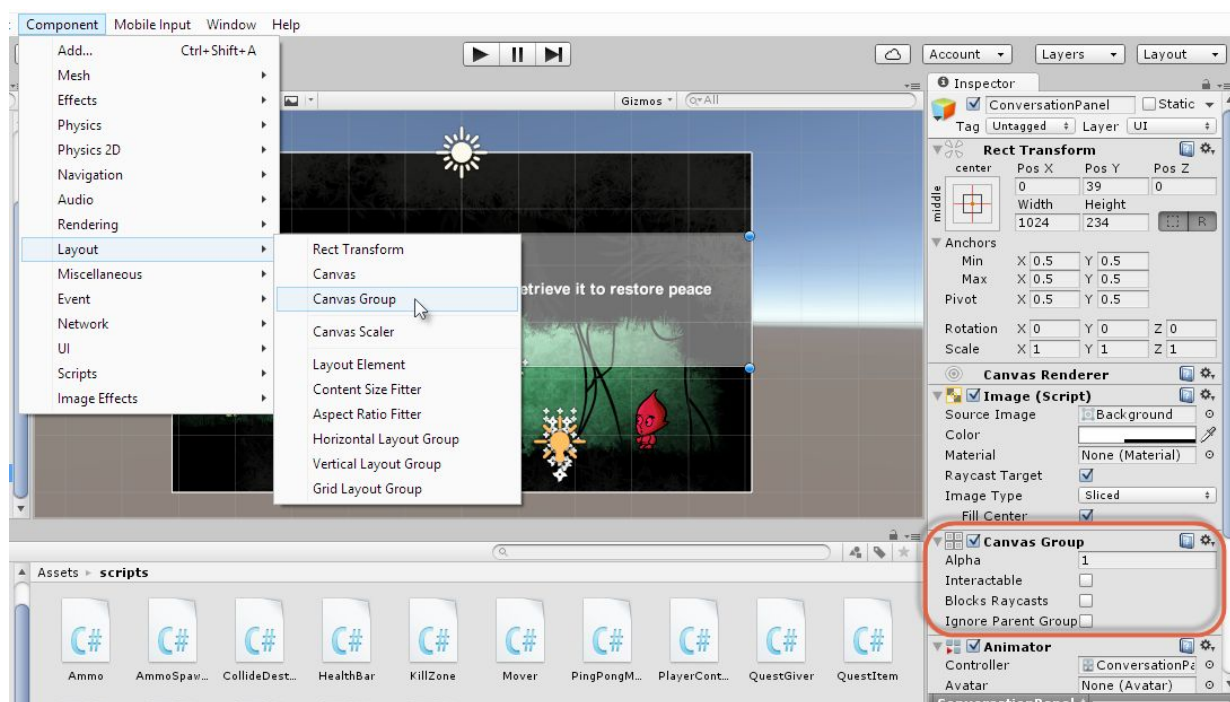


图6.38 向GUI会话面板添加一个画布组组件

现在已经能实现面板的淡入淡出效果了，只需要将Alpha值逐渐由0调整到1即可。不过，仍然需要一个保存任务信息的功能，这样才能确定任务是否已经分配，以及根据任务完成状态确定哪些文本应该显示在对话框中。为了实现这个效果，需要添加一个新的类，将这个类命名为“QuestManager.cs”。这个类可以用来创建和保存任务信息，详见代码示例6.7。

### 代码示例6.7:

```
//-----
using UnityEngine;
using System.Collections;
//-----
[System.Serializable]
public class Quest
{
    //任务完成状态
    public enum QUESTSTATUS {UNASSIGNED=0,ASSIGNED=1,COMPLETE=2};
```

```

        public QUESTSTATUS Status = QUESTSTATUS.UNASSIGNED;
        public string QuestName = string.Empty;
    }
    //-----
    public class QuestManager : MonoBehaviour
    {
        //-----
        //游戏中的所有任务
        public Quest[] Quests;
        private static QuestManager SingletonInstance = null;
        public static QuestManager ThisInstance
        {
            get{
                if(SingletonInstance==null)
                {
                    GameObject QuestObject = new GameObject
("Default");
                    SingletonInstance = QuestObject.
AddComponent<QuestManager>();
                }
                return SingletonInstance;
            }
        }
        //-----
        void Awake()
        {
            //如果这里已经有了一个存在的实例，就销毁游戏对象
            if(SingletonInstance)
            {
                DestroyImmediate(gameObject);
                return;
            }
            //只有一个实例的情况
            SingletonInstance = this;
            DontDestroyOnLoad(gameObject);
        }
        //-----
        public static Quest.QUESTSTATUS GetQuestStatus(string
QuestName)
        {
            foreach(Quest Q in ThisInstance.Quests)
            {
                if(Q.QuestName.Equals(QuestName))
                    return Q.Status;
            }

            return Quest.QUESTSTATUS.UNASSIGNED;
        }

        //-----

```

```

    public static void SetQuestStatus(string QuestName, Quest.
QUESTSTATUS NewStatus)
    {
        foreach(Quest Q in ThisInstance.Quests)
        {
            if(Q.QuestName.Equals(QuestName))
            {
                Q.Status = NewStatus;
                return;
            }
        }
    }
    //-----
    //将任务重置回到未分配的状态
    public static void Reset()
    {
        if(ThisInstance==null)return;

        foreach(Quest Q in ThisInstance.Quests)
            Q.Status = Quest.QUESTSTATUS.UNASSIGNED;
    }
    //-----
}
//-----

```

下面对代码示例6.7进行总结。

- **QuestManager**类中维护了一个包含所有任务的列表，也就是说，这个列表中包含的是游戏中所有的任务，而不仅仅是那些已经分配的或者完成的任务。**Quest**类中定义了一个任务的名字和状态。
- 所有的任务都有一个状态，这个状态可以是以下3种之一：  
**UNASSIGNED**（指还没有分配给玩家的任务），**ASSIGNED**（已经分配给玩家，但是玩家还没有完成），**COMPLETE**（已经分配给玩家，而且玩家已经完成了的任务）。
- 函数**GetQuestStatus**用来检测一个指定任务的完成情况。函数**SetQuestStatus**用来为一个指定任务分配新的状态，这些都是静态函数，因此，任何脚本都可以在任何地方对数据进行获取或者修改。

如果想使用这个对象，需要在场景中（游戏中的第一个场景）创建一个实例，然后在对象检查（Inspector）面板处定义所有可以完成的任务。在游戏中，这里只有一个任务，也就是NPC角色给的任务，在Level2中的危险场景中冒着炮塔的炮火收集宝石。如图6.39所示，这里给出了如何使用“Quest Manager”来配置任务。

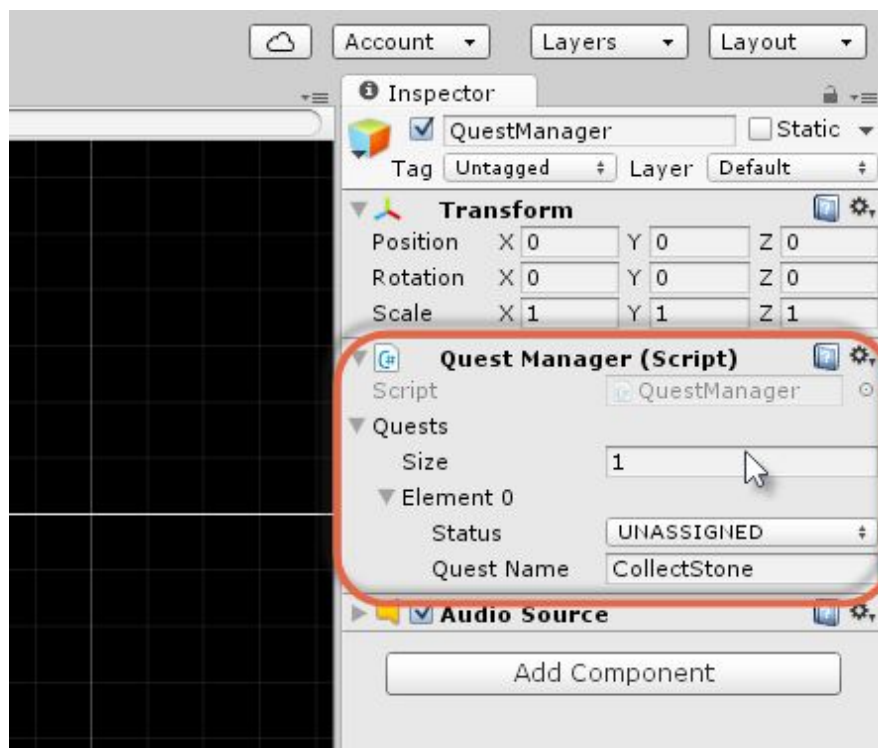


图6.39 使用“Quest Manager”来配置游戏中的任务

“Quest Manager”定义了游戏中所有可能的任务，不管它们是不是已经由玩家领取。不过，NPC仍然需要为玩家分配任务。这一点可以使用脚本来实现，下面编写一个名为“QuestGiver.cs”的脚本。代码示例6.8给出了这个脚本的内容，它应该附加到所有能提供任务的对象上，例如一个NPC。

### 代码示例6.8:

```

//-----
using UnityEngine;
using System.Collections;
using UnityEngine.UI;
//-----
public class QuestGiver : MonoBehaviour
{
    //-----
    //易于理解的任务名称

    public string QuestName = string.Empty;
    //对UI文本盒的引用
    public Text Captions = null;
    //要说的话列表
    public string[] CaptionText;
    //-----
    void OnTriggerEnter2D(Collider2D other)
    {
        if(!other.CompareTag("Player"))return;

        Quest.QUESTSTATUS Status = QuestManager.
GetQuestStatus(QuestName);
        Captions.text = CaptionText[(int) Status]; //Update GUI text
    }
    //-----
    void OnTriggerExit2D(Collider2D other)
    {
        Quest.QUESTSTATUS Status = QuestManager.
GetQuestStatus(QuestName);
        if(Status == Quest.QUESTSTATUS.UNASSIGNED)
            QuestManager.SetQuestStatus(QuestName, Quest.QUESTSTATUS.
ASSIGNED);

        if(Status == Quest.QUESTSTATUS.COMPLETE)
            Application.LoadLevel(5); //Game completed, go to win
screen
    }
}
//-----

```

将这段脚本附加到NPC上之后，就可以开始对这个游戏进行测试了。当玩家走近NPC时，GUI文本应该改变为在对象检查（Inspector）面板中的QuestGiver组件中指定的任务对应的选项。这里的“QuestName”的值应该与QuestManager类中定义的相匹配，如图6.40所示。

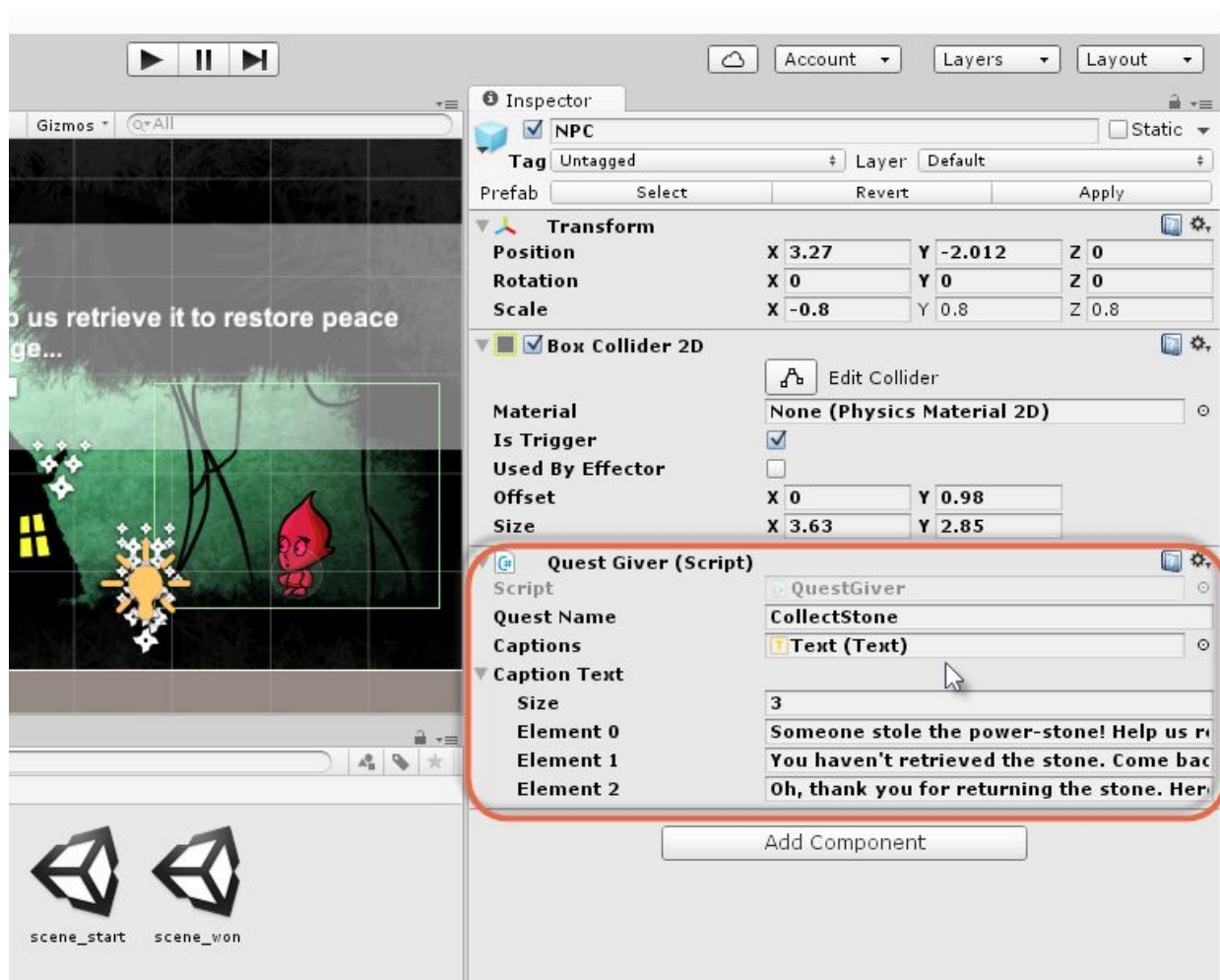


图6.40 定义QuestGiver组件

现在分配的任务是收集宝石，但是关卡中没有宝石，下面添加一个宝石来让玩家收集。将一个宝石贴图从项目面板（贴图文件夹）拖曳到场景2上面的平台上，所以玩家必须爬到上面才能取得宝石（这也是一个挑战），如图6.41所示。给对象添加一个圆形碰撞体（触发器），这样就可以与玩家发生一个碰撞。



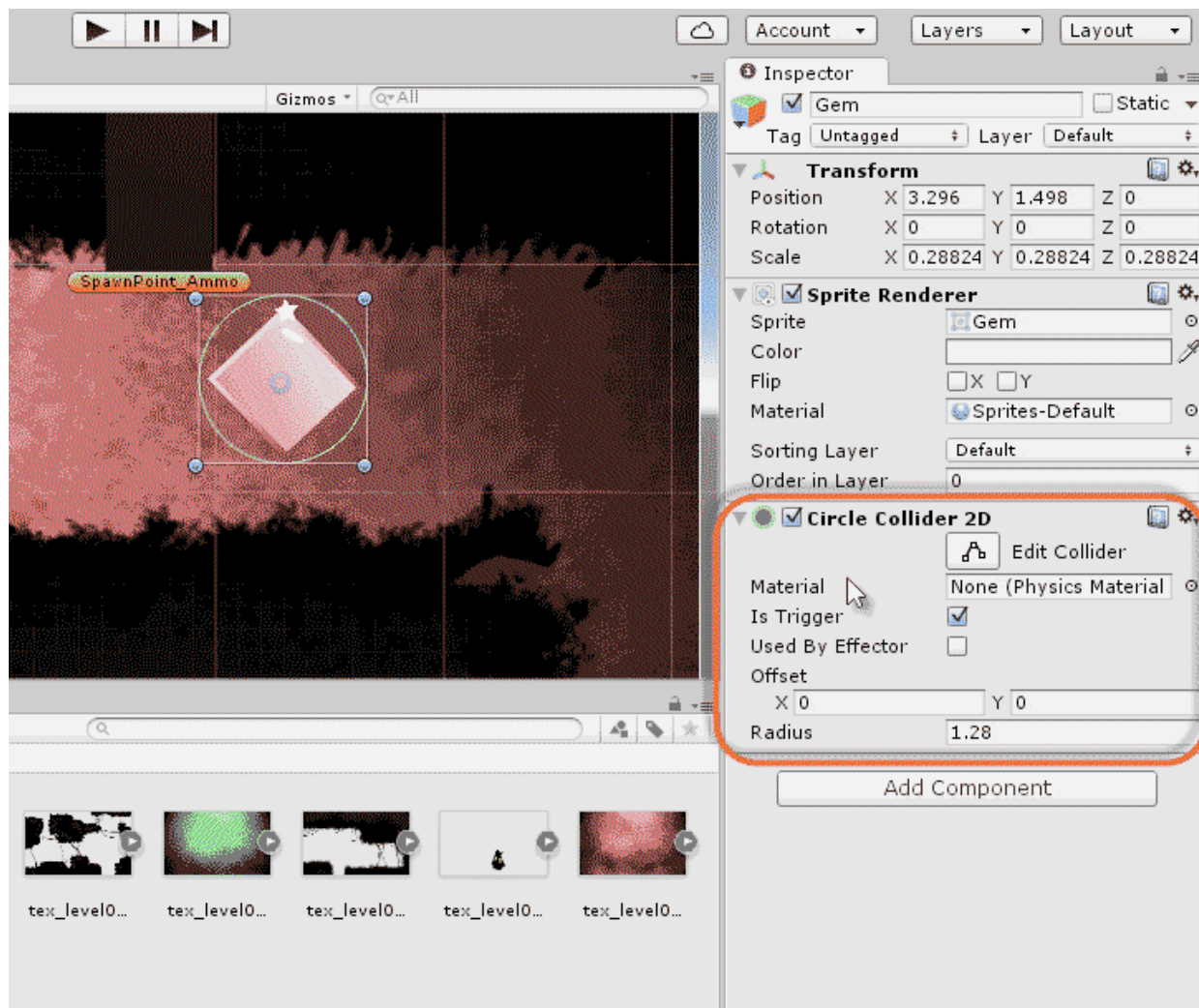


图6.41 创建一个任务对象

最后，需要一段新的QuestItem脚本，当宝石被收集后，这个脚本可以更改Quest Manager类中的任务状态，当玩家再次访问NPC时，允许QuestGiver检测宝石是否已经被收集。QuestItem脚本应该附加到宝石对象上，代码示例6.9是QuestItem脚本的内容。

### 代码示例6.9:

```
//-----
using UnityEngine;
using System.Collections;
```

```

//-----
public class QuestItem : MonoBehaviour
{
    //-----
    public string QuestName;
    private AudioSource ThisAudio = null;
    private SpriteRenderer ThisRenderer = null;
    private Collider2D ThisCollider = null;
    //-----
    void Awake()
    {
        ThisAudio = GetComponent<AudioSource>();
        ThisRenderer = GetComponent<SpriteRenderer>();
        ThisCollider = GetComponent<Collider2D>();
    }
    //-----
    // 初始化函数
    void Start ()
    {
        //隐藏对象
        gameObject.SetActive(false);

        //如果任务已经分配就显示对象
        if(QuestManager.GetQuestStatus(QuestName) == Quest.
QUESTSTATUS.ASSIGNED)
            gameObject.SetActive(true);
    }
    //-----
    //如果物品是可见而且可收集的
    void OnTriggerEnter2D(Collider2D other)
    {
        if(!other.CompareTag("Player"))return;

        if(!gameObject.activeSelf)return;

        //收集成功，任务完成
        QuestManager.SetQuestStatus(QuestName, Quest.QUESTSTATUS.
COMPLETE);

        ThisRenderer.enabled=ThisCollider.enabled=false;

        if(ThisAudio!=null)ThisAudio.Play(); //如果有音效就播放
attached
    }
    //-----
}

```

前面的代码负责在玩家角色进入到了触发区域并将宝石收集之后修改任务的状态，这一过程发生在QuestManager类中。

至此，已经完成了一个完整的任务系统，也设计好了一个NPC角色。这个项目的完整文件可以在本书配套文件的Chapter06/End文件夹中找到。在这里作者强烈建议你对这个文件进行检测，并开始试玩这个游戏，图6.42就给出了游戏进行的界面。

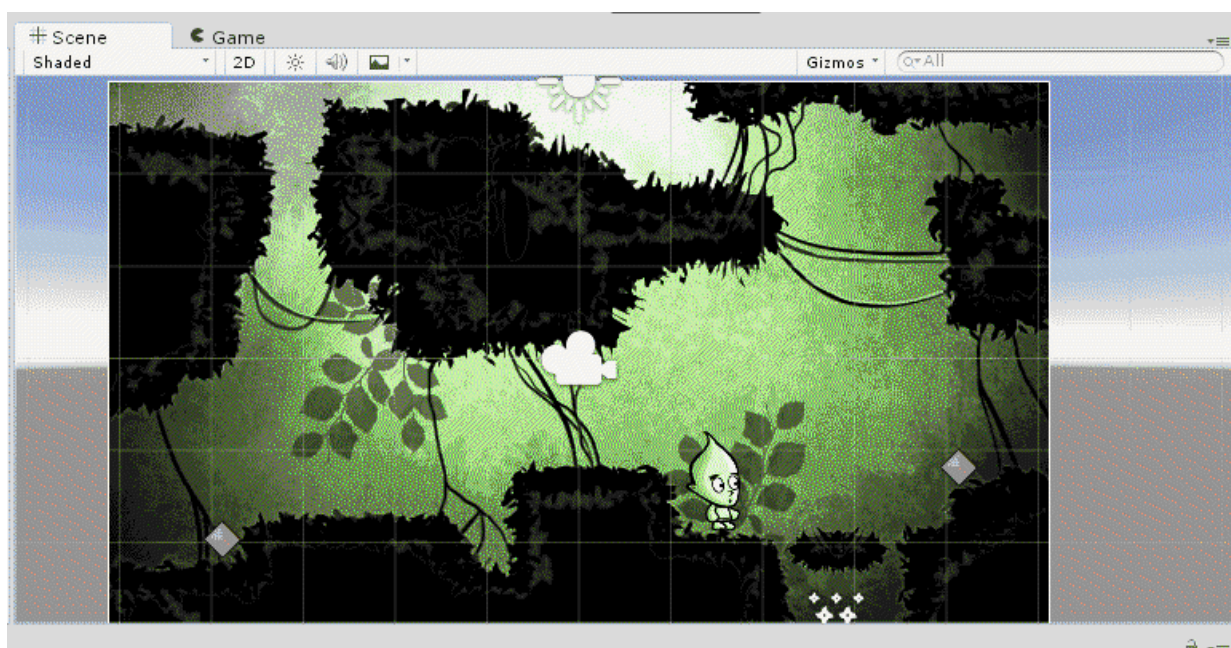


图6.42 已经完成的游戏

## 6.8 小结

很了不起吧！我们现在已经完成了二维冒险游戏。注意一点，为了保证本书的简洁明了，本章中省略了一些小的细节，这些细节在前面的章节已经讲解过。所以，打开教程的文件并对已经完成的项目进行检测，查看代码是如何工作的这一点十分重要。本书到现在为止已

经完成了3个Unity项目。所以，以前所有的内容都已经结束，在下一章中将迎来最后的大结局——第四个项目。

## 第7章 有智慧的敌人 (I)

这一章将创建最后一个项目，这个项目中包含一个广阔的地面。不同于之前的3个项目，这个项目将不会彻底完结，也没有明确的胜利和失败条件，只是一个用来实现功能原型和概念验证的项目，在游戏中突出了一系列的重要编码技术和思想。具体来说，创建一个具有地形的世界、第一人称视角角色以及一些敌人。这些敌人都拥有人工智能，它们在场景中进行巡逻并搜索玩家，如果发现玩家就会发起攻击。在这一章中，将就以下主题进行探讨：

- 如何使用地形工具来构建关卡和地形
- 如何生成以及使用导航网格
- 如何为人工智能的开发做准备



### 注意

本章所需要的项目文件和资源可以在本书的配套文件Chapter07/Start文件夹中找到，如果没有建立自己的项目，那么就可以使用这里面的文件。

### 7.1 项目概览

要创建的项目是一个第一人称视角游戏，游戏实现了一个第一视角的玩家角色原型在地形环境中进行漫游和探索。这个地形中包括了丘陵、沟谷以及其他地形。在这个起伏不平的地形中还分布着一些敌

人的角色（NPC），每个敌人都有自己的人工智能。具体来说，每个敌人都是四处游荡（巡逻模式）去搜索玩家。如果玩家进入了敌人的视线内，NPC将会开始追逐玩家（追捕模式）。如果在追逐过程中，玩家逃出了NPC的视线，它们将重新开始巡逻模式。另一方面，如果敌人在追逐过程中接近了玩家，那么它们就会对玩家发起攻击（攻击模式）。所有的AI（人工智能）都具有3个状态：巡逻模式、追捕模式和攻击模式。这就构成了敌人的AI，也是在这个项目中对玩家的主要威胁。图7.1所示为已经完成的完整项目。

图7.1 构建一个包含了具有智能NPC的游戏世界

## 7.2 入门指南

仍然以创建一个新的项目作为本章的开始，这个过程在以前的章节中已经有过详细的介绍。在整个项目中，将会使用到3个Unity内置的资源包，即角色（Characters）资源包、特效（Effect）资源包、环境（Environment）资源包。这些资源包可以通过单击应用程序菜单上的“Assets | Import Packages”来实现，如图7.2所示。

图7.2 导入资源包

首先创建游戏世界（陆地），这个世界是一个室外环境，即一个由草原、丘陵和山脉共同构成的游戏世界。可以使用3D建模软件比如说3DS MAX、Maya或者Blender来创建一个这样的景观，再将这样的景



观导入到Unity中。不过，在Unity中内置了地形设计工具，虽然在很多方面（如即将看到的）还很有限，但是仍然不失为一款强大而通用的工具。在应用程序菜单处依次选择“GameObject | 3D Object | Terrain”来创建一个新的地形，如图7.3所示。

图7.3 创建一个新的地形

在成功完成创建操作之后，一个地形（Terrain）对象就被添加到了场景中的原点（0,0,0）位置。注意，由于这个地形的大小原因，它可能不会立刻出现在视窗里。接下来解决这个问题，首先在层次面板中选中场景中的地形对象，然后按下键盘上的“F”键，这样就可以让这个对象处于视图的中心位置。这个对象看起来就像是一个平面对象，如图7.4所示。但是与平面对象不同的是，可以改变这个对象的形状，也可以改变它的造型，这一点接下来就会看到。

图7.4 向场景中添加一个地形对象

在对地形进行形状和造型的改变之前，应该在对象检查（Inspector）面板处进行一些初始的地理设置，以确保这个地理结构支持所需要的地形类型，而且大小也合适。首先在视图中选中这个地形对象，然后在对象检查（Inspector）面板上单击齿轮图标，这样就可以显示出地形的设置，如图7.5所示。

图7.5 查看并编辑地形的设置

默认情况下，针对大多数的目的地形都显得有些太大（500m × 500m）了，现在将其缩小为256 × 256，可以调整的更小。具体的操作是，将“Width”和“Length”的值都设置成256。Height值表示整个地形中所有区域中的顶点或者山峰所能到达的最大高度。出于优化的角度来考虑，地形的大小够用就可以了，无需做得过大，因为地形对象往往是高度细化并且性能密集的，如图7.6所示。在对地面的造型进行改变之前，首先要确认对地形的面积进行设置，以避免做一些无用功。

图7.6 设置地形分辨率的宽度（Width）和长度（Length）

## 7.3 地形的构建

现在来塑造地形。首先选择好地形对象，然后在对象检查（Inspector）面板处单击地形组件最左边的选项板图标（这是抬高和降低地形的工具）。这样就可以选择各种刷子形状来绘制地形细节。选择一个软的圆形大刷（可以使用刷大小的滑动条来调节大小），并使用透明度来设置刷子的强度，然后在地形上单击并拖动这个刷子来绘制地形的细节。在地形上创建一些丘陵和山脉，如图7.7所示。记住，如果需要，就按下“Shift”键，这样当单击时就会反转（降低）地形。

图7.7 查看和编辑地形设置

如果现在的地形看起来太过粗糙，显得不够自然，可以使用抚平高度（**Smooth Height**）工具将地面上的细节进行平滑处理，具体的操作是，按下**Terrain**组件上的第三个按钮，如图7.8所示。当选择了这个工具之后，就可以跟以前一样设置刷子的形状、大小以及透明度，当地形上单击之后，就可以实现地形高度变化的平滑。

图7.8 启用抚平高度（**Smooth Height**）工具

现在已经设置好了地面的形状和造型，而且也对其进行了平滑处理，接下来该给它上色了。目前的地形是灰色的、沉闷的、没有进行任何修饰的。整个地形没有明确的贴图或者景物，例如草或者岩石。现在使用“**Paint Texture**”工具来修正这个问题，在对象检查

（**Inspector**）面板上的“**Terrain**”组件处单击“**Paint Texture**”按钮（第四个按钮）。如果是第一次执行这个操作，就需要准备和加载一组用于着色的贴图，如图7.9所示。

图7.9 为地形着色准备贴图

单击“**Edit Textures**”按钮，在出现的上下文菜单中选中“**Add Texture...**”选项，然后就会出现一个贴图配置对话框，允许在选项板上添加新的贴图，如图7.10所示。

图7.10 向选项板中添加贴图

可以使用项目面板来查找本地的Unity环境资源包地形贴图，在“Texture Selection”对话框打开并准备加载第一个贴图。这些地图可以在“Standard Assets | Environment | TerrainAssets | SurfaceTextures”文件夹中找到。基于这个案例的特点，选择一个长满青草的贴图，它将被用来作为一个基础贴图来填充整个地面。从项目面板中将长满青草的贴图拖曳到“Texture Selection”对话框的“Albedo”槽位中，后面的“Normal”槽位保持为空即可，如图7.11所示。

图7.11 选择一个基础贴图

当向“Texture Selection”对话框中添加完第一个贴图之后，一定要设置贴图的大小。这里指的是贴图的大小（单位为米），一个单独的贴图覆盖的面积。想要获得一个准确的平铺值，需要一个不断尝试的过程，需要不断地进行设置、预览、再调整，直到看起来合适。对于这个例子，使用的值为 $75 \times 75$ ，然后单击添加按钮，如图7.12所示。

图7.12 设置贴图平铺尺寸

当单击“Add”按钮之后，基本贴图就会平铺在地面上。从远处看，这些场景视图中平铺的贴图显得十分明显，而且看起来也很不舒服。你可能希望在此基础上对平铺设置重新进行调试。不过从一个第一人称视角来看，效果就完全不一样了。基于这个原因，使用一个第一人称控制器预设体（从内置的资源包），以第一人称的视角来预览这个地形，看看这些贴图是如何平铺在地面上的，如图7.13所示。

图7.13 预览地面上平铺的贴图

如果需要对现在的贴图平铺选项进行编辑，可以在对象检查（Inspector）面板上选中“Terrain”组件中贴图选项板中的贴图预览图，然后选择“Edit Textures”按钮，如图7.10所示。

在现在这个阶段，地形对象由一个长满青草的贴图作为基础贴图，无缝地平铺在整个表面。虽然看起来很棒了，但是如果地形中包含更多的贴图，例如草、岩石，甚至沙漠风格的地表，形态就更好了（见图7.14）。这一点可以通过向地形选择对话框中添加更多的贴图来实现，只需单击“Edit Textures”按钮，从弹出的上下文菜单中选中“Add Texture”。然后将一个新的不同贴图拖动到贴图选择对话框中的Albedo槽位中，重复这个过程，将所有需要的贴图都添加进来。当关闭这个对话框时，所有添加进来的贴图就都会显示在对象检查（Inspector）面板上的贴图选项板（“Textures Palette”）处。

图7.14 将贴图添加到贴图工具板上

分配给绘图刷的贴图会在对象检查（Inspector）面板里以高亮的蓝色边框表示。可以通过单击缩略图来选择不同的贴图，当这样操作时，选中的贴图就被应用到了绘图刷上，可以通过单击它来应用到地形上。单击绘图刷并在地形上拖动，就可以在上面喷绘贴图。还可以设置“Brush Shape”“Brush Size”“Opacity”“Target Strength”的值来控制应用贴图的程度，以及如何将其融合到地形表面，如图7.15所示。

图7.15 分层绘制和贴图融合

继续完成地形的绘制，直到创建一个赏心悦目的外观和感觉。地形绘制完成以后，在层次面板上选中场景中的定向光源（**Directional Light**），调整它的“**Rotation**”属性使它如同场景中的太阳一样。请注意，可通过将光源完成一个**360度**的旋转来模拟一个完整的昼夜周期（就照明和外观而言）。因此，可以通过使用动画窗口来轻松为游戏模拟出黑夜和白昼的交替，正如前面的章节中介绍过的一样，如图7.16所示。

图7.16 完成的地形

最后使用第一人称控制器资源来对地形进行探索。按下工具栏上的“**Play**”图标，开始整个关卡的环游。现在已经拥有了一个包含地形的游戏世界了，如图7.17所示。

图7.17 第一人称视角进行地形探索

在开始新的内容之前，先来考虑一下Unity地形工具的技术限制，以及这可能对游戏产生的影响。具体而言，Unity地形是一种基于高度的地图，这意味着地形的高度（起伏）是基于图像文件（高度图）中的灰度像素来产生的。当使用检查（**Inspector**）面板上的笔刷绘制地形时，实际上是在高度图上绘制用来映射地形的像素点。这是一个聪明



又有趣的方法，但是这种方法有一个重要的限制，高度图实际上是一个二维地形贴图，这样就导致了Unity地形并不是真正意义上的三维，它并不包含洞穴、裂缝、溶洞或者任何的裂口，玩家不能进入到地面之下。高度图由上下两个部分组成，两个部分都没有内部空间。在许多情况下，这都并不会引起问题。但是有时可能会需要使用内部空间，如果这样做，就会需要一个Unity自带地形系统的替代软件，可以是手工三维软件（例如3DS Max、Maya和Blender等），也可以是资源商店里的各种插件。

## 7.4 导航与导航网格

整个游戏世界的地形已经完成了，现在必须开始考虑整个项目的主要目标了。具体来说，这个关卡应该是一个人工智能实验，要创造可以自由地在地形漫游的敌方NPC人物，当玩家进入他们的视野时就会追逐和攻击玩家。为了实现这个目标，必须要适当地配置寻路操作。当考虑到NPC人工智能和NPC在关卡中的运动时，就必须要考虑到关卡中地形的崎岖，在这个地形中充满了丘陵、山脉、斜坡以及坑洼等问题。如果一个NPC角色想要在这个地形中顺利漫游，就不得不考虑到很多复杂的问题。例如，NPC就不能简单地沿着直线从A点直接到达B点，因为这样直接过去，有可能会穿过一些实体对象或者地形。NPC角色需要像一个有智慧的人一样，会绕过障碍物，可以上坡，也可以下坡，这一点对于创造出一个真实的角色是十分重要的。为NPC寻找合适路径的计算过程被称为寻路操作，让玩家角色按照这些找到的路径移动的过程被称为导航。Unity中内置了寻路和导航功能，所以在为NPC计算路径和导航时就变得简单了很多。

为此，必须生成导航网格。导航网格是场景中的一个特殊的网格资源，它是一条不进行渲染的几何曲线，这条曲线沿着地形表面。使用这个导航网格来实现对一个角色移动的寻路和导航操作，可以在应用程序菜单处选中“**Window | Navigation**”来产生一个导航网格，如图7.18所示。

图7.18 访问导航窗口

导航窗口的目的是生成一个实际上贴近地面（**Floor**）层的低逼真度的地形网格。为了让这个过程正确工作，场景中的所有不可以移动的“**Floor**”网格都必须标注为导航静态（**Navigation Static**），只需要在层次面板处选中地形，然后在对象检查（**Inspector**）面板处单击“**Static**”下拉列表框，然后启用“**Navigation Static**”选项，如图7.19所示。

图7.19 将不能移动的“**Floor**”对象设置为静态

现在来访问“**Navigation**”窗口（通常停放在“**Inspector**”面板的旁边），单击“**Bake**”选项卡访问主要的“**Navigation**”设置。在这个选项卡中，可以控制一系列影响导航网格生成的设置选项，如图7.20所示。

图7.20 **Bake**选项卡中包含了生成导航网格的主要设置

生成一个初始的导航网格作为开始，以此来查看系统默认设置下导航网格的样子。如果需要，可以使用新的设置来轻易地清除和再次生成导航网格。首先在对象检查（Inspector）面板处单击“Bake”按钮，当单击完成后，一个默认的导航网格就产生了，它会在场景视图中以蓝颜色显示出来，如图7.21所示。

图7.21 默认的导航网格

不过默认的导航网格是有问题的，它表示关卡中全部可以行走的区域。所以从本质上来说，它是NPC们被限制只能在其中行走，而不能超出的范围。从上面的图像可以看出，这个导航网格在许多地方断裂和破碎，有些区域与其他区域隔离和断开了。这一点通常是不可取的，因为这将意味着所有的NPC都只能行走在一个孤立的区域，不能走到另一个区域，因为两个区域之间没有连通的供NPC行走的导航网格。要修正这个问题，必须对两个设置进行调整。第一个就是“Agent Radius”设置，它表示代理（NPC）的平均大小，会直接影响到导航网格在地面网格上展开时的宽度。将“Agent Radius”的值减小，然后再一次单击“Bake”按钮来查看结果，如图7.22所示。

图7.22 使用代理半径（Agent Radius）来重新定义网格

通过这个操作明显改善了网格，但是现在地图仍然有一些断裂和破碎的区域。这是由“Max Slope”设置造成的，它用来控制当表面的陡峭程度（例如山的斜坡）为多大时，对于一个NPC来说是不可以行走

的，增大这个值就可以将导航网格变得更大，然后单击“**Bake**”按钮，如图7.23所示。

图7.23 将NPC所能翻越的最大坡度增加，以扩大导航网格

现在已经在关卡中创建好了一个导航网格。这个导航网格资源将被保存在一个与场景名称相匹配的文件夹中。当在项目面板中选中这个资源时，就可以对导航网格中的各种只读属性，例如“**Height**”“**Walkable Radius**”等进行预览，如图7.24所示。

图7.24 从项目面板中对各种导航网格属性进行预览

## 7.5 构建一个NPC

现在开始构建一个能展示人工智能的NPC角色。首先要使用Unity本身自带资源中的“**Ethan**”网格。这个资源可以在项目面板上的“**Standard Assets | Characters | ThirdPerson Character | Models**”文件夹中找到，将“**Ethan**”网格模型拖动到场景中，然后将其放置在地面上。对这个模型进行细化和编辑，最终使用它创建一个预设体来表示一个NPC角色，如图7.25所示。

图7.25 开始创建一个NPC角色

当向关卡中添加了一个**Ethan**模型之后，要确保角色的蓝色前向向量的确指向前方，面向着角色实际上正在看着的方向。如果这个前向向量没有前对齐，可以创建一个空对象，然后将它作为一个模型的子对象与角色模型对齐，这样作为父对象的角色模型的前向向量就会沿着角色模型的视线向前。也就是说，蓝色的前向向量应该与角色模型眼睛注视的方向对齐（与视线对齐），如果希望这个角色走路时有真实感，那么这一点十分重要，如图7.26所示。



图7.26 位于角色脚部的指向前方的前向向量（蓝色箭头）

NPC角色应该可以利用导航网格来智能地在关卡中行走。为此，应该为角色附加一个导航网格代理组件。在关卡中选中“**Ethan**”，然后在应用程序菜单选择“**Component | Navigation NavMesh Agent**”。导航网格代理（**NavMeshAgent**）组件中包括了寻路和引路（导航）功能，可以帮助一个游戏对象在导航网格中移动，如图7.27所示。

图7.27 将一个导航网格代理组件附加到NPC上

默认情况下，导航网络会为每个代理，也就是导航和行走的对象，分配一个圆柱碰撞体。这不是一个具有物理行为的真碰撞体，而是一个用来检测代理是否走近导航网格边缘的伪碰撞体。选中使用Ethan创建的NPC，在对象检查（Inspector）面板里的“Nav Mesh Agent”组件处将“Height”设置为1.66，并将“Radius”设置为0.22，这样可以使碰撞体与网格对象的体型更加接近，如图7.28所示。

图7.28 重新设置代理碰撞器

移动网格对象进行测试，查看是否一切都能正常工作。对此，需要创建一个新的脚本。首先，创建一个新的空对象，它将作为一个目的地，也就是一个NPC应该到达的目标位置，具体操作是在应用程序菜单中依次选中“Game Object | Create Empty”，将这个对象命名为“Destination”。然后为它分配一个小图标，这是为了使它在视图中可见，如图7.29所示。当这个对象处于选中状态时，在对象检查（Inspector）面板单击左上方的方块图标，然后在显示出来的图标中选择想要的。

图7.29 创建一个目标对象



接下来，创建一个新的C#脚本文件（FollowDestination.cs），然后将这个脚本添加到场景中的NPC对象。所有的代码都包含在代码示例7.1中。

### 代码示例7.1:

```
using UnityEngine;
using System.Collections;

public class FollowDestination : MonoBehaviour
{
    private NavMeshAgent ThisAgent = null;
    public Transform Destination = null;

    // 初始化函数
    void Awake ()
    {
        ThisAgent = GetComponent<NavMeshAgent>();
    }
    // Update函数会在每一帧调用一次
    void Update ()
    {
        ThisAgent.SetDestination(Destination.position);
    }
}
```

下面对代码7.1示例进行总结。

- FollowDestination类可以附加到任何具有导航网格代理组件的对象上。这个对象在移动时应该跟随目标对象。
- Destination变量中保存了要跟随的目标对象。

当将脚本附加到NPC对象上之后，就将目标空对象拖曳到对象检查（Inspector）面板里的“FollowDestination”组件处的“Destination”槽位处，这样就会为脚本设定一个目的地，如图7.30所示。

图7.30 配置一个跟随目标的对象

现在对游戏进行一个测试。在游戏期间，在场景选项卡中移动目标物体并查看NPC的反应，会发现NPC不断地追逐着目标对象。此外，如果在对象检查（Inspector）面板中打开的导航窗口中游戏，并且在层次面板中选中了NPC，场景视图中就会显示一些诊断信息选项，通过这些选项就可以实现NPC计算路线的可视化，如图7.31所示。

图7.31 对NPC导航进行测试

## 7.6 创建巡逻的NPC

现在已经有有了一个可以跟随目标对象移动的NPC了，这是一个很有意义的练习，但是我们仍需要比这更复杂的功能。具体来说，需要NPC能巡逻，也就是说，要通过一个路径点的系统来到达多个目的地，按照顺序从一个目的地到另一个目的地。有很多种方法来实现这个目标，一种方法就是通过脚本实现，利用这种方法创建一系列的路径点对象，然后循环遍历这些对象，就是当NPC到达一个对象之后，再移动到下一个对象。这个方法是相当有效的，不过这里还有另一种选择。具体而言，就是不使用脚本，创建一个动画来将一个单独的目标对象随着时间移动到不同的路径点，因为NPC会一直跟随着这个不断移动的目标，所以也就实现了在关卡中不断的巡逻。

现在采取第二种方法，首先在应用程序菜单处依次单击“Window | Animation”打开动画窗口，如图7.32所示。如果愿意，出于便于浏览的

目的，可以将动画窗口以横向视图的方式显示在项目面板上。

图7.32 访问动画窗口

接下来，从层次面板上选中要做成动画的对象（目标对象），然后在动画窗口中，单击“**Create**”按钮。在这里，系统将会询问是否保存这个动画，以及要为此动画起个名字，这里为此动画命名为“**anim\_DestPatrol**”，如图7.33所示。

图7.33 创建一个新的动画

当动画创建完成以后，就可以继续定义动画频道了。对于目标对象，需要为对象的位置字段添加一个频道，因为对象的位置在场景中是变化的。在动画窗口中单击“**Add Property**”按钮，然后依次单击“**Transform | Position**”来添加一个新的位置频道，这样就会自动在时间线上创建起始的关键帧，同时也对应着对象的位置，如图7.34所示。

图7.34 创建一个新的动画

单击动画窗口时间线上那条红色的竖线，并拖曳它到0和1之间，然后将场景选项卡中的目标对象移动到一个新的位置。当完成这个操作之后，Unity就会自动记下目标对象在这个关键帧时的位置。在时间线上不断重复这个过程，并在每次都移动目标对象到不同的位置，这样就会创建一个完整的巡逻动画出来，如图7.35所示。

图7.35 创建一个巡逻动画

在动画窗口或者工具栏上按下“**Play**”按钮播放动画。默认情况下，动画的播放速度太快（正如我们即将看到的，这个问题很容易解决），但是也要注意，目标对象正如我们预期的一样，逐渐改变它的位置。也就是Unity动画引擎会自动在时间轴上关键帧之间插入动画，使得目标对象在路径点之间平滑的移动。不过，现在只希望看到目标如何在路径点之间传送，不需要其中的任何过渡，这就需要修改动画曲线（**Animation Curve**）中的差值模式（**Interpolation Mode**）。单击动画窗口左下方的“**Curves**”按钮。默认情况下，动画窗口是处于“**DopeSheet**”模式，这种模式下可以轻易地看到关键帧并改变它们。而“**Curve**”模式可以修改关键帧之间的插值，如图7.36所示。

图7.36 访问动画曲线

现在，框选择（单击并拖动选择框）图表视图中所有的关键帧，然后单击鼠标右键，会显示出关键帧的上下文菜单，在菜单中选择“**Right Tangent | Constant**”，这时所有关键帧的句柄都变成了菱形，这表示所有关键帧的值都会一直保持不变直到下一帧为止，如图7.37所示。

图7.37 修改关键帧的句柄

当在菜单上选中“Constant”选项之后，关键帧之间的曲线看起来就会完全不同，现在是直线来连接两个点，如图7.38所示。

图7.38 常数差值

按下工具条上的“Play”按钮，之后目标就会按照动画的进展在路径点上跳跃前进，NPC会不断地向着目标移动和前行。由于动画的默认速度，NPC可能会对迅速改变位置的目标感到困惑，显得无所适从。为了解决这个问题，在层次面板上选中目标对象，双击动画组件的控制器区域，打开附加在对象上的动画图，它控制在何时执行指定的动画，如图7.39所示。

图7.39 访问动画资源

你也可以在应用程序菜单手动的选择“Window |Animator”，在动画窗口中，默认节点以橘黄色高亮显示。这个节点将会在关卡中的对象第一次启动时播放，如图7.40所示。

图7.40 “DestPatrol”动画是动画窗口中的默认动画

选中图中的“DestPatrol”节点，然后在对象检查（Inspector）面板中减小“Speed”的值。在本案例中，使用了0.2作为“Speed”的值，这样游戏工作起来很顺利。当对这个值进行修改时，一定要记得再对游戏进行测试来查看它的效果，如图7.41所示。

图7.41 减小动画的速度

当按下“Play”按钮时，NPC就会在多个目的地之间以一个看起来比较真实的速度从一个路径点移动到另一个路径点。如果NPC在路径点之间移动得太快或者太慢，就可以通过增加或者减少速度来获得所需要的结果。现在已经拥有了一个完整的动画路径点系统，如图7.42所示。

图7.42 运作中的路径点系统

## 7.7 小结

干得不错，我们现在已经完成了第一个人工智能的第一部分：建立了一个地形，产生了一个导航网格，并创建了一个基本的路径点系统，角色可以在目的地之间移动。这是模拟人工智能的一个很好的开始，但是如果想要达到预期的效果，还需要做更多添加代码方面的工作。这些内容将集中在下一章也就是最后一章中进行。



## 第8章 有智慧的敌人 (II)

本章是最后一章，将会继续上一章的内容，专注于创建一个有智慧敌人的相关理论和编码技术。这些敌人将会表现出3种主要行为：巡逻、追击和进攻。本章将深入讨论以下主题：

- 如何对敌人的人工智能系统进行设计和编码
- 如何编码有限状态机
- 如何创建视野范围功能



本章所需要的项目文件和资源可以在本书的配套文件的Chapter08/Start文件夹中找到，如果没有建立自己的项目，就可以使用其中的文件来开始本章。

### 8.1 敌人的人工智能——视野范围

现在根据游戏的功能要求来开发敌人的人工智能。场景中的敌人将会以一个巡逻模式开始，从一个地方徘徊到另一个地方不断地搜索玩家角色。如果敌人发现了玩家角色，就会改变巡逻并开始追逐，试图接近玩家并发起攻击。如果玩家进入了敌人的攻击范围内，敌人将会从追逐变为攻击。如果玩家成功地摆脱了敌人，那么敌人就会停止

追逐并再次开始巡逻，就像它们一开始那样。以上描述了我们所需要的敌人的人工智能行为。

为了实现这些功能，需要对敌人的视野范围功能进行编码，敌人需要能够看到玩家角色，或者时刻检测玩家是否在敌人的视线中，这将帮助敌人决定它们应该继续巡逻，还是开始追逐玩家角色。实现这个功能最好使用源文件的LineSight.cs脚本。下面的代码示例8.1中的脚本文件应该添加到上一章中创建的敌人角色身上。

### 代码示例8.1:

```
using UnityEngine;
using System.Collections;
//-----
public class LineSight : MonoBehaviour
{
    //-----
    //我们视野的敏感度
    public enum SightSensitivity {STRICT, LOOSE};

    //瞄准具灵敏度
    public SightSensitivity Sensitivity = SightSensitivity.STRICT;

    //我们能看到目标吗?
    public bool CanSeeTarget = false;

    //视野
    public float FieldOfView = 45f;

    //对目标的引用
    private Transform Target = null;

    //对眼睛的引用
    public Transform EyePoint = null;

    //对“transform”组件的引用
    private Transform ThisTransform = null;

    //对球状碰撞体的引用
    private SphereCollider ThisCollider = null;
```

```

//对最后对象视野的引用
public Vector3 LastKnowSighting = Vector3.zero;
//-----
void Awake()
{
    ThisTransform = GetComponent<Transform>();
    ThisCollider = GetComponent<SphereCollider>();
    LastKnowSighting = ThisTransform.position;
    Target = GameObject.FindGameObjectWithTag("Player").
GetComponent<Transform>();
}
//-----
bool InFOV()
{
    //获取到目标的方向
    Vector3 DirToTarget = Target.position - EyePoint.position;

    //获取正前方和目标之间的角度
    float Angle = Vector3.Angle(EyePoint.forward, DirToTarget);

    //我们在视野中吗?
    if(Angle <= FieldOfView)
        return true;

    //不在视野中
    return false;
}
//-----
bool ClearLineofSight()
{
    RaycastHit Info;

    if(Physics.Raycast(EyePoint.position, (Target.position -
EyePoint.
position).normalized, out Info, ThisCollider.radius))
    {
        //如果看到Player了
        if(Info.transform.CompareTag("Player"))
            return true;
    }

    return false;
}
//-----
void UpdateSight()
{
    switch(Sensitivity)
    {
        case SightSensitivity.STRICT:
            CanSeeTarget = InFOV() && ClearLineofSight();
            break;
    }
}

```

```

        case SightSensitivity.LOOSE:
            CanSeeTarget = InFOV() || ClearLineofSight();
            break;
        }
    }
    //-----
    void OnTriggerStay(Collider Other)
    {
        UpdateSight();

        //更新最后的视野
        if(CanSeeTarget)
            LastKnowSighting = Target.position;
    }
    //-----
    void OnTriggerExit(Collider Other)
    {
        if(!Other.CompareTag("Player"))return;

        CanSeeTarget = false;
    }
    //-----
}
//-----

```

下面对代码进行几点总结。

- **LineSight**类应该附加到所有的敌人角色对象上，目的是为了计算在玩家和敌人之间是否存在一个可用的直视视线。
- 变量**CanSeeTarget**是一个**Boolean**类型（**true/false**），这个值会在每一帧更新一次，描述敌人现在是否可以看见玩家（当前帧）。“**true**”表示玩家在敌人的视野中，“**false**”表示玩家不在敌人的视野中。
- 变量**Field of View**是一个浮点数值，它表示敌人眼睛所能看到的两侧之间的角度，在这个范围内的对象（如玩家）是可以看到的。这个值设定得越大，敌人发现玩家的概率就越大。

- **InFOV**函数将会返回**true**或者**false**，这个值表示玩家是否处于敌人的视野范围内。但是这个值忽略了玩家是否被墙或者其他实体（如支柱）挡住的情况，它只是简单地考虑了敌人眼睛的位置，确定一个到玩家的向量，并测量前向向量和玩家之间的角度。然后将这个值与变量**FieldOfView**进行比较，如果玩家与敌人之间的角度小于变量**Field of View**，就返回**true**。简而言之，如果有一个明确的视线，这个函数可以告诉你敌人是否可以看见玩家。
- 函数**ClearLineOfSight**返回值也是**true**或者**false**，这个值表示在敌人的眼睛和玩家之间是否存在任何物理的障碍（碰撞体），例如墙或者道具。这个函数并不考虑玩家是否在敌人的视野范围内。将这个函数与**InFOV**函数结合使用，就可以判断玩家是否处在敌人的无遮挡的视野范围内，因此可以认为玩家是可见的。
- 函数**OnTriggerStay** 和**OnTriggerExit**这两个函数分别表示玩家进入到了敌人的触发区域和玩家离开了敌人的触发区域。正如我们所看到的，一个球状的碰撞器可以附加到敌人角色身上代表它的视野。这意味着在一定距离或者半径内，敌人可以看见里面的玩家，只要视野清晰且没有遮挡即可。

现在将**LineSight.cs**脚本附加到场景中的敌人角色上，将球形碰撞体组件（标记为触发器）设置成近似于敌人的视野范围大小，如图8.1所示，保留“**Field of View**”的值为45，这个值可以增加，如果需要，设置到90左右，以此来调整敌人的视野范围。

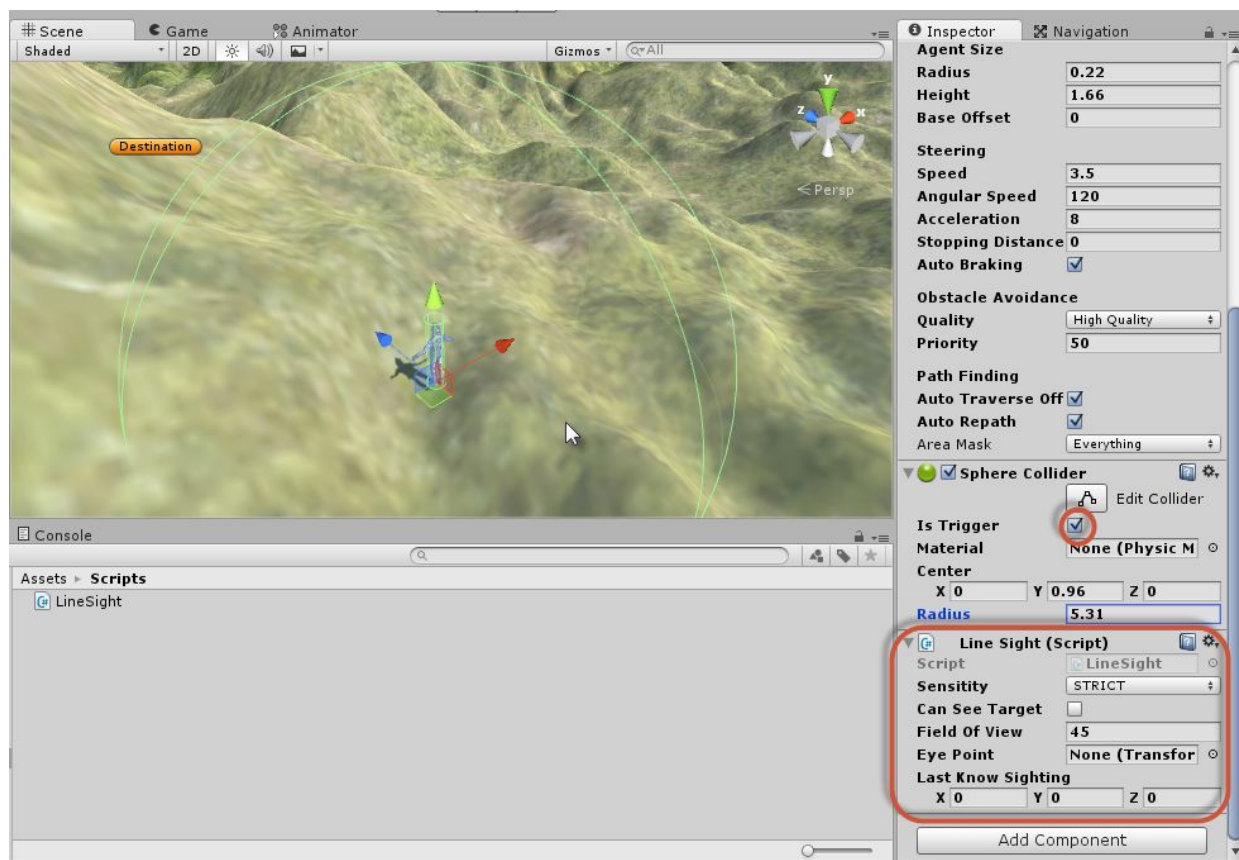


图8.1 向NPC添加一个视野范围

“Eye Point”的值默认设置为“None”，表示一个空值。这应该指向一个充当敌人角色视点的特定位置，这个位置就是敌人角色眼睛的位置。可以利用应用程序菜单来创建一个新的空游戏对象，方法是依次单击“GameObject | CreateEmpty”，将这个对象命名为“Eye Point”。在对象检查（Inspector）面板中使用“Gizmo”图标来激活它的可见性，然后把它添加到敌人对象上作为一个子对象。接着将这个对象定位到角色的视点位置上，确认前向向量朝向同一方向，如图8.2所示。



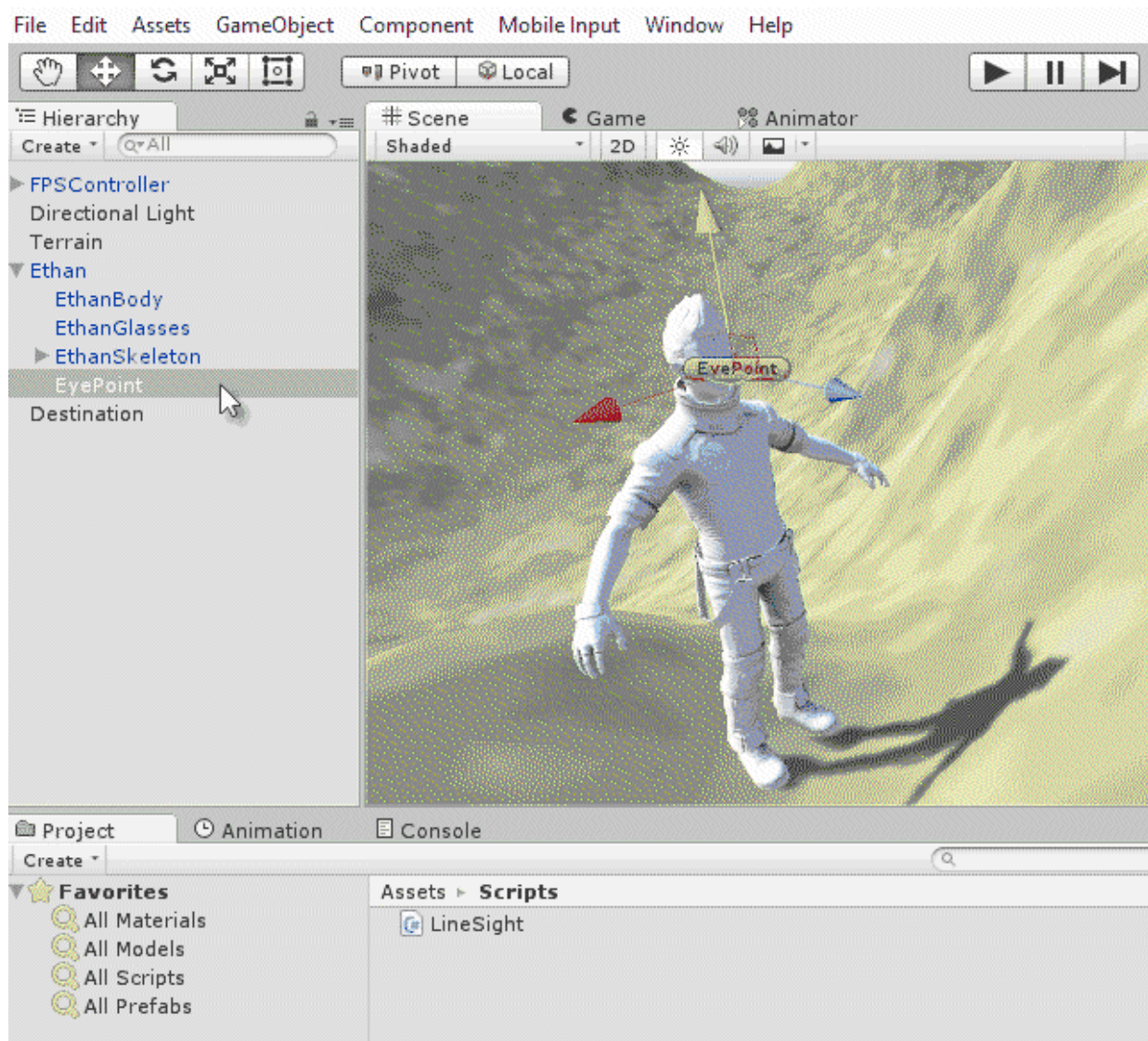


图8.2 添加一个视点

现在从“Hierarchy”面板上将“Eye Point”对象都拖曳到对象检查（Inspector）面板中的“LineSight”组件处的“Eye Point”后面的槽位里。这样操作之后，“Eye Point”对象就成为了敌人的视点。利用这个视点可以判断敌人是否能看到玩家。使用这样一个单独的视点而不是使用敌人角色的位置是十分有效的，因为敌人角色的位置实际上是敌人脚的位置，而不是眼睛的位置，如图8.3所示。

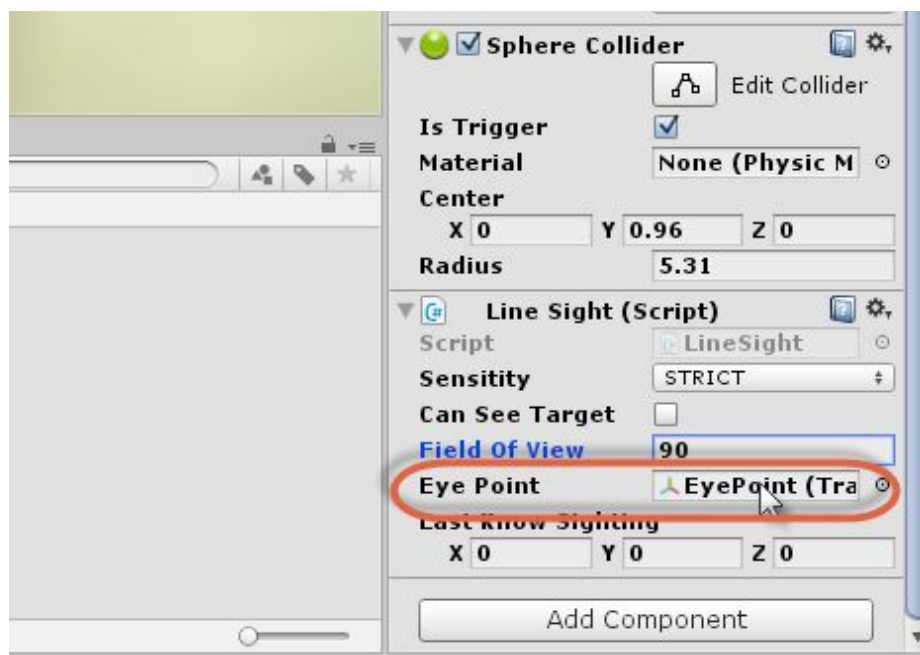


图8.3 为NPC定义一个视点

最后，脚本LineSight会通过Player标签来在场景中寻找Player对象，从而对Player定位。因此，必须要确保玩家对象上使用了Player标签，如图8.4所示。

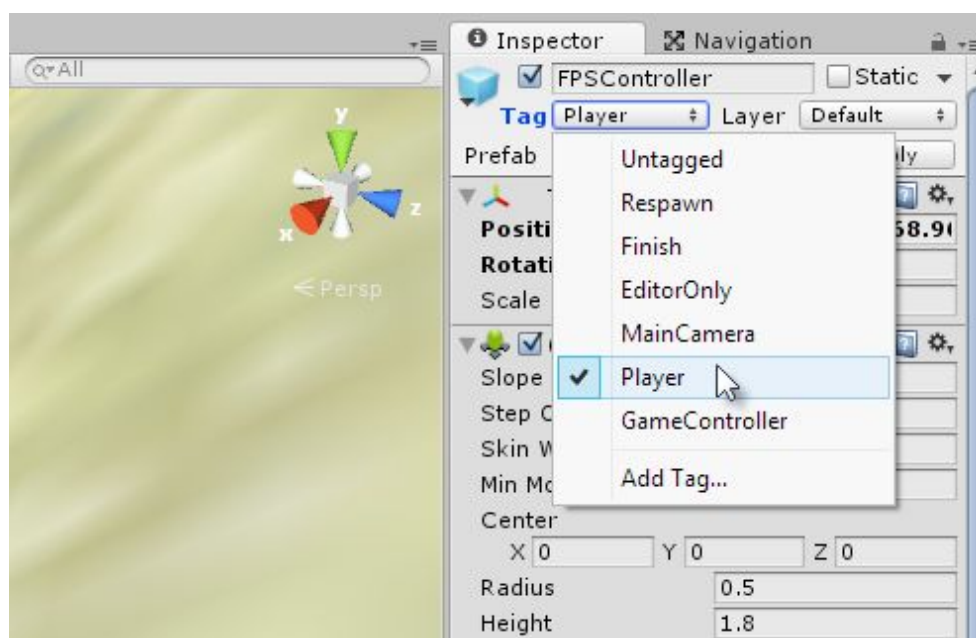


图8.4 为玩家对象打上Player标签

现在来测试一下游戏，当接近NPC对象时，“Can See Target”区域就会被启用，如图8.5所示。干得不错，视线功能已经完成了，让我们继续前进！

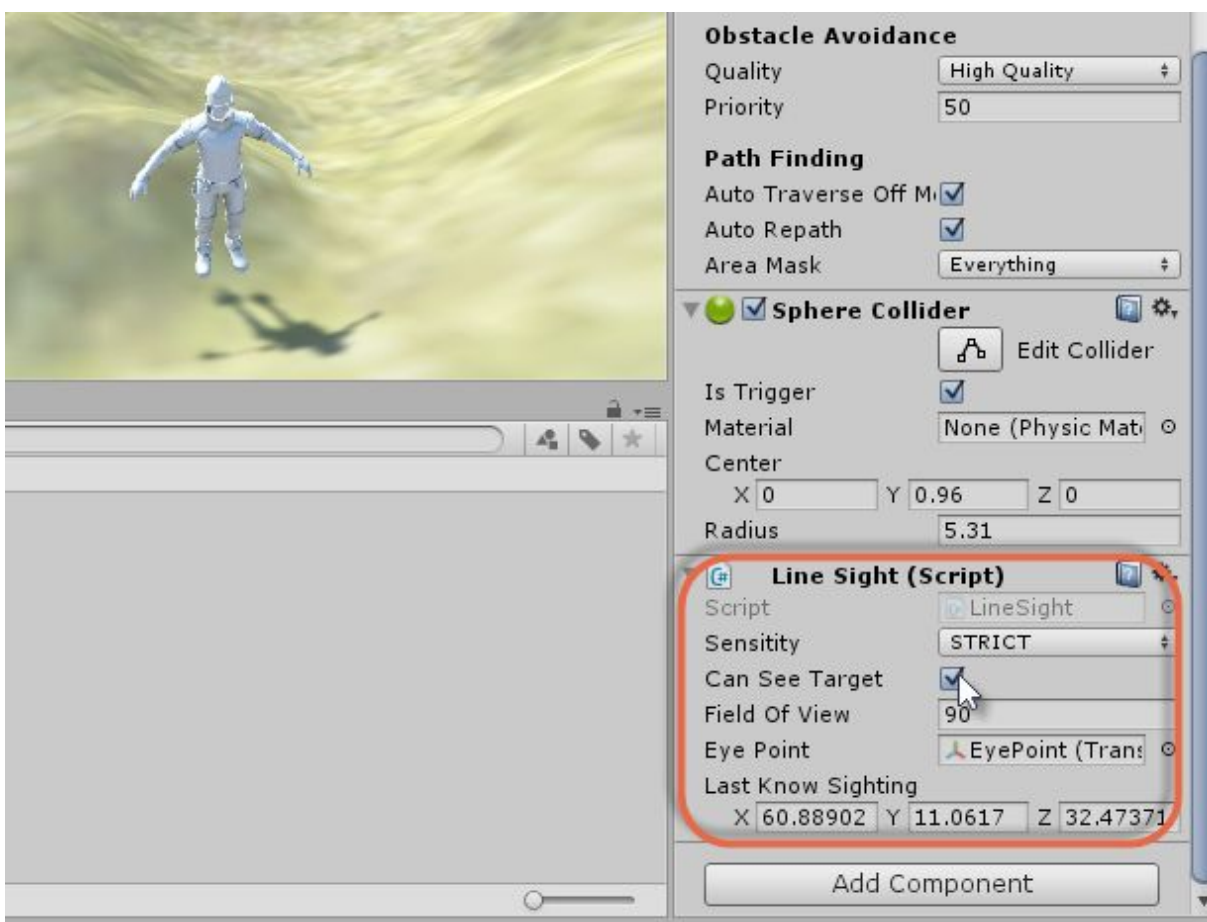


图8.5 对视线功能进行测试

## 8.2 有限状态机概述

如果为NPC对象创建一个AI（人工智能），刚刚创建的视线功能还不够，还需要使用有限状态机（Finite State Machines, FSMs）。有

限状态机并不是Unity中的内容，也不是C#语言的组成部分。有限状态机是一个概念、框架或者说是想法，将有限状态机应用在编码中可以实现特定的人工智能行为。它来自于智能角色的一种具体思维方式。我们总结了一下NPC在任意时刻可能处于的全部3种状态，分别是巡逻模式（当敌人没有目标而徘徊时）、追逐模式（当敌人在追赶玩家时）以及攻击模式（当敌人已经接近玩家并开始攻击时）。这些模式是一种具有独特行为而且唯一的状态，敌人在任何时间都必须且只能处于这3种状态之一。例如，敌人不能同时巡逻和追逐，也不能同时巡逻和攻击，因为这不符合游戏中设计的逻辑。

除了这些状态之外，还有一个状态连接的规则集，其决定了一个状态在何时转换成另一个状态。例如，NPC可以看到玩家但是并不具备攻击条件的时候，它只能从巡逻状态切换到追逐状态。同样，如果敌人在攻击状态时失去玩家的踪影，它们应该从进攻切换到巡逻状态。因此，这些状态和它们的连接规则的组合形成了一个有限状态机，任何实现了这个功能的代码其实就是一个有限状态机。编码实现有限状态机的方法本身并没有什么正确或者错误之分，这里有很多简单方法，对于特定的目的来说，其中有一些是合适的，另一些是不合适的。这一节将使用Coroutines来实现有限状态机的编码。从创建一个主要结构开始，代码示例8.2给出的AI\_Enemy.cs脚本中的代码就是这样的一个主要结构：

### 代码示例8.2:

```
using UnityEngine;
using System.Collections;
//-----
public class AI_Enemy : MonoBehaviour
{
```

```

//-----
public enum ENEMY_STATE {PATROL, CHASE, ATTACK};
//-----
public ENEMY_STATE CurrentState
{
    get{return currentstate;}

    set
    {
        //更新当前状态
        currentstate = value;

        //停止当前运行的所有coroutines
        StopAllCoroutines();

        switch(currentstate)
        {
            case ENEMY_STATE.PATROL:
                StartCoroutine(AIPatrol());
                break;

            case ENEMY_STATE.CHASE:
                StartCoroutine(AIChase());
                break;

            case ENEMY_STATE.ATTACK:
                StartCoroutine(AIAttack());
                break;
        }
    }
}
//-----
[SerializeField]
private ENEMY_STATE currentstate = ENEMY_STATE.PATROL;

//对视线组件的引用
private LineSight ThisLineSight = null;

//对导航网格代理的引用
private NavMeshAgent ThisAgent = null;

//对玩家transform的引用
private Transform PlayerTransform = null;

//-----
void Awake()
{
    ThisLineSight = GetComponent<LineSight>();
    ThisAgent = GetComponent<NavMeshAgent>();
    PlayerTransform = GameObject.FindGameObjectWithTag("Player").
    GetComponent<Transform>();
}

```

```

}
//-----
void Start()
{
    //配置起始状态
    CurrentState = ENEMY_STATE.PATROL;
}
//-----
public IEnumerator AIPatrol()
{
    yield break;
}

//-----
public IEnumerator AIChase()
{
    yield break;
}
//-----
public IEnumerator AIAttack()
{
    yield break;
}
//-----
}
//-----

```



关于Coroutines的更多信息可以在Unity的在线文档  
<http://docs.unity3d.com/Manual/Coroutines.html> 处查看。

下面对代码示例中的内容进行几点总结。

- **AI\_Enemy**类创建到现在并没有完全地实现一个完整的有限状态机，而只是实现了一个开始的框架。它说明了整体的结构，其中每一个状态对应一个单独的coroutine。



- **CurrentState**变量定义了当前选中的激活状态，终止所有当前的coroutine并启动相关的coroutine。
- 每个状态coroutine将会运行在一个框架安全（Frame-safe）的无限循环中，只要有限状态机是活动的，这个循环就不停止。这将允许敌人对象更新它的行为，下面很快会看到。

在下一步开始之前，要确保将AI\_Energy脚本附加到了NPC对象上，如图8.6所示。

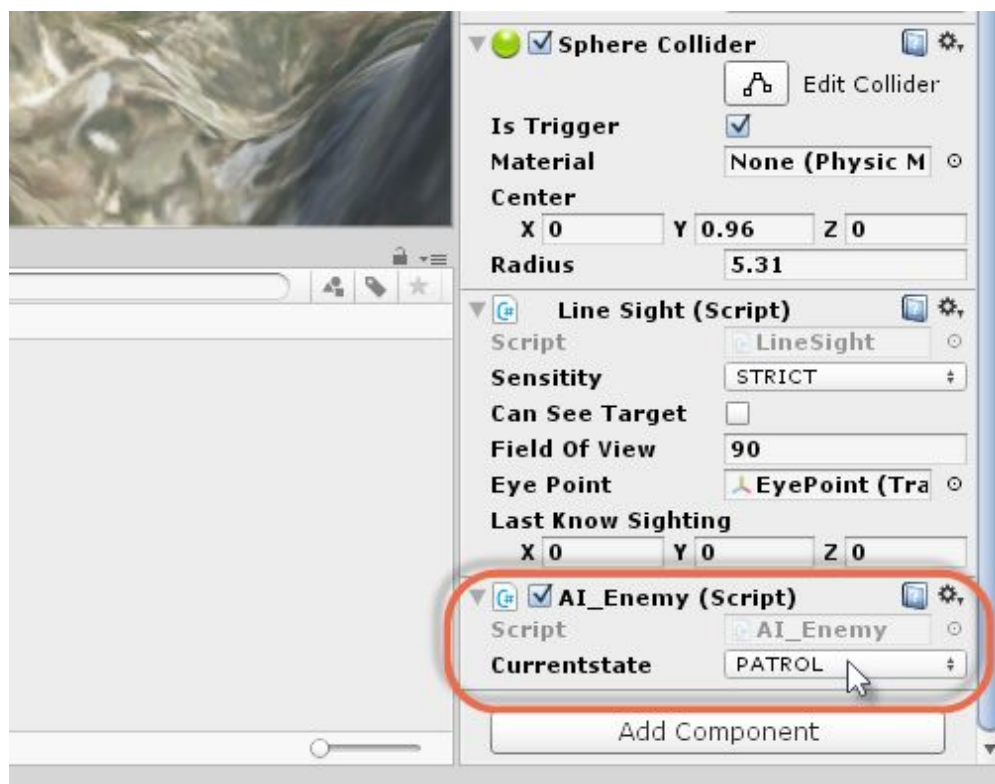


图8.6 将AI\_Energy脚本附加到NPC角色上

## 8.3 巡逻状态

NPC人工智能3个状态中的第一个状态是巡逻状态。在前面的章节中，已经配置了动画的巡逻对象，NPC处在巡逻状态时，会一直跟着这些巡逻对象移动。巡逻对象会根据预先定义好的动画资源从一个位置移动到另一个位置，不断地在场景中巡逻。不过，在此之前，NPC只是简单地跟随这个对象没有尽头，而巡逻状态需要NPC都能看见玩家是否在它的路线上。如果看见，状态就需要改变。为了支持这个功能，需要对巡逻状态和AI\_Energy的Start函数进行编码，下面的代码示例8.3出了具体的代码内容。

### 代码示例8.3:

```
void Start()
{
    //获得一个随机的目的地
    GameObject[] Destinations = GameObject.
FindGameObjectsWithTag("Dest");
    PatrolDestination = Destinations[Random.Range(0, Destinations.
Length)].GetComponent<Transform>();

    //配置起始状态

    CurrentState = ENEMY_STATE.PATROL;
}
//-----
public IEnumerator AIPatrol()
{
    //在巡逻的时候一直循环
    while(currentstate == ENEMY_STATE.PATROL)
    {
        //设置搜索为STRICT
        ThisLineSight.Sensitivity = LineSight.SightSensitivity.STRICT;

        //追逐到巡逻位置
        ThisAgent.Resume();
        ThisAgent.SetDestination(PatrolDestination.position);

        //等待直到路径计算完成
        while(ThisAgent.pathPending)
            yield return null;

        //当可以看到目标的时候就启动追逐模式
```

```
    if(ThisLineSight.CanSeeTarget)
    {
        ThisAgent.Stop();
        CurrentState = ENEMY_STATE.CHASE;
        yield break;
    }

    //Wait until next frame
    yield return null;
}
}
```

下面对代码示例8.3进行几点总结。

- **Start**函数将敌人的初始状态设置为巡逻模式，**CoroutineAIPatrol**操作这个模式。
- **AIPatrol Coroutine**在巡逻状态激活时会一直循环。记住，在一个**Coroutine**与一个**yield**组合下，无限循环不一定是坏事，这可以实现对长期行为进行简单整齐的编码。
- **SetDestination**函数用来调用将**NavMeshAgent**发送到指定目的地。后面跟随一个**pathPending**控制，它是**NavMeshAgent**中的一个变量。这个控制会一直持续到**pathPending**的值变为**false**的时候，此时意味着已经计算完了从源地址到目的地的整个路径。对于简单的或者短的旅程，路径几乎可以立刻计算出来，但是对于复杂的路径，可能需要很长的时间。
- 在巡逻状态时，需要不断地对**LineSight**组件进行检查，以便知道敌人是否具有到玩家的一个直视视线，如果具有，那么敌人立刻由巡逻状态转为追逐状态。
- 记住，**yield**返回的空语句将会暂停一个**Coroutine**直到下一帧。

现在，将**AIEnemy**脚本拖曳到场景中的**NPC**角色上。巡逻模式下**NPC**将跟随一个移动的物体而运动，也就是说敌人会一直跟随一个移

动的目的地。在上一章中已经使用**Animation**窗口在场景中创建了一个随着时间移动的对象，这个对象会从一个地方跳到另一个地方，如图8.7所示。

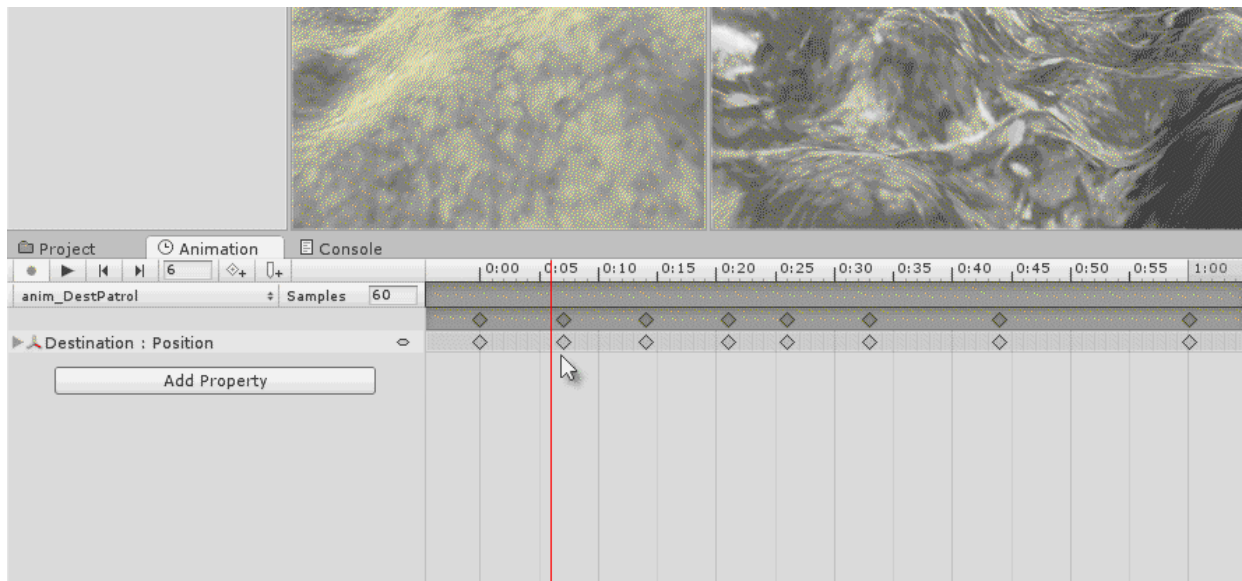


图8.7 创建可移动的对象

为了获得可移动的对象，可以在场景中创建一个或者更多的目的地对象，将它们的“Tag”属性设置为“Dest”。AIEnemy对象的Start函数会搜索所有场景Tag属性为“Dest”的对象，这些对象将会被当作目的地点，如图8.8所示。

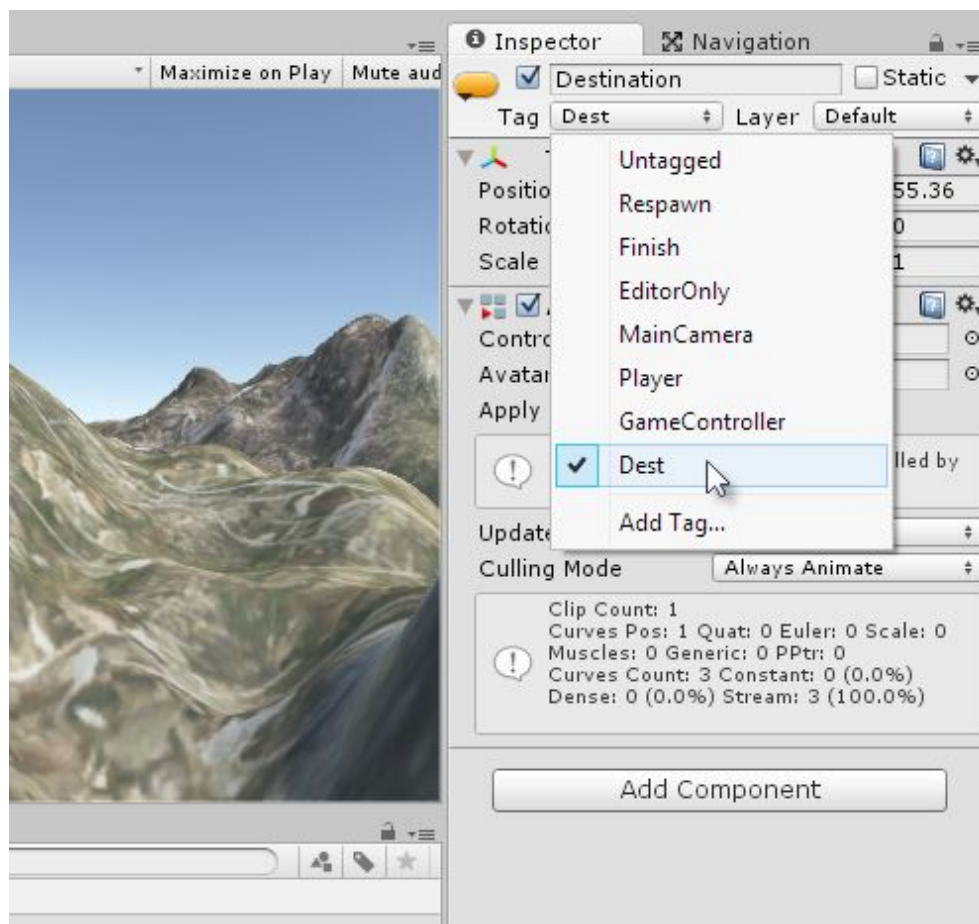


图8.8 对目的地对象进行标记

## 8.4 追逐状态

追逐状态是敌人有限状态机3种状态中的第二种，这个状态直接关联到巡逻和攻击状态，它可以转换到两种状态之一。如果一个正在巡逻的NPC建立了一个到玩家的直接视线，那么NPC就会由巡逻模式转为追逐模式。相反，如果一个正在攻击的NPC落在了玩家的范围之外（也许因为玩家逃跑了），NPC会再次转入追逐模式。从追逐状态本身，它可以转换成巡逻状态或者攻击状态，相反，就会进入追逐状

态。如果NPC与玩家的距离接近到了可以攻击的距离内，就会切换到攻击模式，下面的代码示例8.4进行了修改，实现了追逐行为。

#### 代码示例8.4:

```
public IEnumerator AChase()
{
    //循环追逐状态
    while(currentstate == ENEMY_STATE.CHASE)
    {
        //将查找模式设置为“loose”
        ThisLineSight.Sensitivity = LineSight.SightSensitivity.LOOSE;

        //追踪的最后一个位置
        ThisAgent.Resume();
        ThisAgent.SetDestination(ThisLineSight.LastKnownSighting);

        //等待直到路径计算完成
        while(ThisAgent.pathPending)
            yield return null;
        //我们是否到达目的地?
        if(ThisAgent.remainingDistance <= ThisAgent.stoppingDistance)
        {
            //停止代理
            ThisAgent.Stop();

            //到达目的地但是无法看到玩家
            if(!ThisLineSight.CanSeeTarget)
                CurrentState = ENEMY_STATE.PATROL;
            else //Reached destination and can see player. Reached
                attacking distance
                CurrentState = ENEMY_STATE.ATTACK;

            yield break;
        }
        //等待到下一帧
        yield return null;
    }
}
```

下面对代码示例8.4进行几点总结。



- 当进入到追逐状态以后，就会启动AIChasecoroutine，跟巡逻状态一样，只要它处于激活状态，Coroutine就会无限循环。
- NavMeshAgent的成员变量remainingDistance用来检测NPC是否进入了攻击玩家的范围。
- LineSight类中的CanSeeTarget布尔型变量表示玩家是否可见，并会影响到NPC是否返回到巡逻模式。

现在来运行一下这段代码，此时已经拥有了一个可以巡逻和追逐的敌人角色。

## 8.5 攻击状态

第三个也是最后一个NPC的状态是攻击状态，在此期间NPC会不断地攻击玩家。只能从追逐状态转换到攻击状态。在追逐过程中，NPC必须检测是否进入了攻击距离，如果进入了，就必须将追逐状态切换到攻击状态。在攻击状态中，如果玩家远离NPC并超出了攻击距离，NPC就必须从攻击状态转换为追逐模式，下面的代码示例8.5包含了EnemyAI类及全部的代码。

### 代码示例8.5:

```
using UnityEngine;
using System.Collections;
//-----
public class AI_Enemy : MonoBehaviour
{
    //-----
    public enum ENEMY_STATE {PATROL, CHASE, ATTACK};
    //-----
    public ENEMY_STATE CurrentState
    {
        get{return currentstate;}
```

```

set
{
    //更新当前状态
    currentstate = value;

    //停止所有正在运行的coroutines
    StopAllCoroutines();

    switch(currentstate)
    {
        case ENEMY_STATE.PATROL:
            StartCoroutine(AIPatrol());
            break;

        case ENEMY_STATE.CHASE:
            StartCoroutine(AIChase());
            break;

        case ENEMY_STATE.ATTACK:
            StartCoroutine(AIAttack());
            break;
    }
}
}
//-----
[SerializeField]
private ENEMY_STATE currentstate = ENEMY_STATE.PATROL;

//对视线组件的引用
private LineSight ThisLineSight = null;
//对导航网络代理的引用
private NavMeshAgent ThisAgent = null;

//对玩家生命值的引用
private Health PlayerHealth = null;

//对玩家transform的引用
private Transform PlayerTransform = null;

//对巡逻目的的引用
private Transform PatrolDestination = null;

//每秒钟的伤害值
public float MaxDamage = 10f;
//-----
void Awake()
{
    ThisLineSight = GetComponent<LineSight>();
    ThisAgent = GetComponent<NavMeshAgent>();
}

```

```

    PlayerHealth = GameObject.FindGameObjectWithTag("Player").
GetComponent<Health>();
    PlayerTransform = PlayerHealth.GetComponent<Transform>();
}
//-----
void Start()
{
    //获取随机的目的地
    GameObject[] Destinations = GameObject.
FindGameObjectsWithTag("Dest");

    PatrolDestination = Destinations[Random.Range(0, Destinations.
Length)].GetComponent<Transform>();
    //配置初始状态
    CurrentState = ENEMY_STATE.PATROL;
}
//-----
public IEnumerator AIPatrol()
{
    //巡逻的时候不断循环
    while(currentstate == ENEMY_STATE.PATROL)
    {
        //将查找模式设置为"strict"
        ThisLineSight.Sensitivity = LineSight.SightSensitivity.STRICT;
        //追逐到巡逻位置
        ThisAgent.Resume();
        ThisAgent.SetDestination(PatrolDestination.position);

        //等待路径计算完成
        while(ThisAgent.pathPending)
            yield return null;

        //如果我们看到目标就开始追逐
        if(ThisLineSight.CanSeeTarget)
        {
            ThisAgent.Stop();
            CurrentState = ENEMY_STATE.CHASE;
            yield break;
        }

        //等待直到到下一帧
        yield return null;
    }
}
//-----
public IEnumerator AIChase()
{
    // 追逐的时候不断循环
    while(currentstate == ENEMY_STATE.CHASE)
    {

```

```

//将查找模式设置为loose
ThisLineSight.Sensitivity = LineSight.SightSensitivity.LOOSE;

//追逐到最后一个目的地
ThisAgent.Resume();
ThisAgent.SetDestination(ThisLineSight.LastKnowSighting);

//等待路径计算完成
while(ThisAgent.pathPending)
    yield return null;

//我们是否到达目的地?
if(ThisAgent.remainingDistance <= ThisAgent.stoppingDistance)
{
    //停止代理
    ThisAgent.Stop();

    //到达目的地但是看不到玩家
    if(!ThisLineSight.CanSeeTarget)
        CurrentState = ENEMY_STATE.PATROL;
    else //Reached destination and can see player. Reached
attacking distance
        CurrentState = ENEMY_STATE.ATTACK;

    yield break;
}

//等待直到到下一帧
yield return null;
}
}
//-----
public IEnumerator AIAttack()
{
    //在追逐和攻击的时候循环
    while(currentstate == ENEMY_STATE.ATTACK)
    {
        //等待路径计算完成
        ThisAgent.Resume();
        ThisAgent.SetDestination(PlayerTransform.position);

        //Wait until path is computed
        while(ThisAgent.pathPending)
            yield return null;

        //玩家是不是已经跑了?
        if(ThisAgent.remainingDistance > ThisAgent.stoppingDistance)
        {
            //改变回追逐模式
            CurrentState = ENEMY_STATE.CHASE;

```

```

        yield break;
    }
    else
    {
        //攻击
        PlayerHealth.HealthPoints -= MaxDamage * Time.deltaTime;
    }

    //等待直到到下一帧
    yield return null;
}
yield break;
}
//-----
}
//-----

```

下面对代码示例8.5进行几点总结。

- 当攻击状态（敌人处于这个状态之后会进行攻击）被激活以后，**AIAttack coroutine**就会无限制地执行。
- 变量**MaxDamage**用来指明敌人每秒会对玩家造成的伤害。
- **AIAttack coroutine**依靠**Health**来造成对生命值的伤害。这是一个自定义的生命值编码组件。玩家和敌人都应该拥有一个表示他们健康情况的生命值组件。

脚本**Health**（**Health.cs**）由**AIEnemy**类所引用，对玩家造成伤害。基于这个原因，玩家需要附加一个生命值组件，这个组件的代码包含在下面的代码示例8.6中。

### 代码示例8.6:

```

using UnityEngine;
using System.Collections;

public class Health : MonoBehaviour
{

```

```

public float HealthPoints
{
    get{return healthPoints;}
    set
    {
        healthPoints = value;

        //如果生命值小于0，就会死亡
        if(healthPoints <= 0)
            Destroy(gameObject);
    }
}

[SerializeField]
private float healthPoints = 100f;
}

```

**Health**脚本十分简单，它负责维护一个数值形式的生命值，当这个生命值下降到0或者0以下时，就会摧毁这个脚本附加的对象。这个脚本必须附加到玩家角色上，允许NPC靠近玩家时对其造成伤害。同样，这个脚本也可以附加到NPC对象上，允许玩家对NPC进行攻击，如图8.9所示。

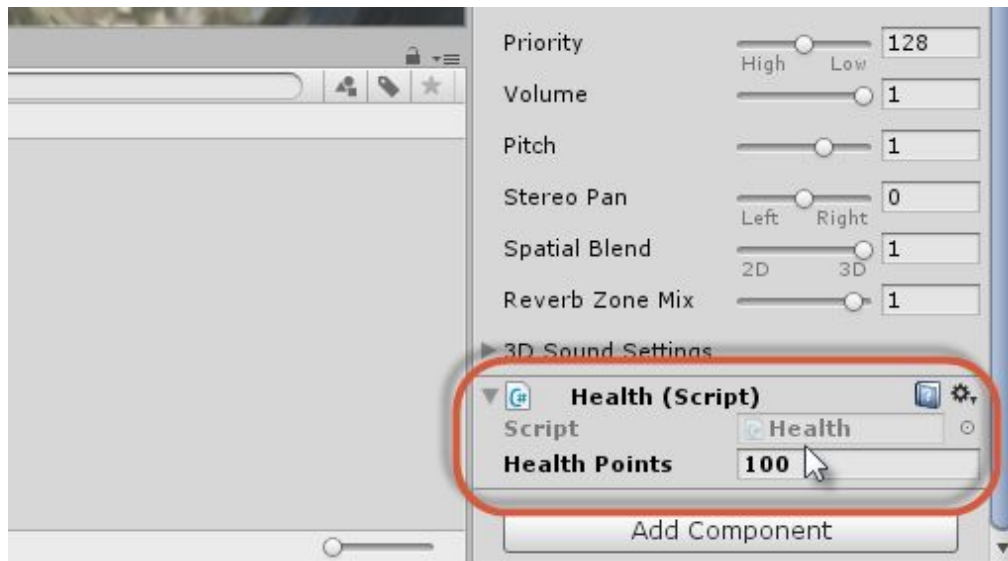


图8.9 配置玩家的生命值



现在已经做好了对这个项目的测试准备。首先，把敌人对象做成一个预设体，具体的方法就是将NPC游戏对象从场景视图或者层次（Hierarchy）面板拖动到项目面板上，然后就可以在场景中创建任意数量的敌人了，如图8.10所示。



图8.10 创建一个NPC预设体

按下工具栏上“Play”图标来测试这个关卡。现在已经开发好了一个完整的环境，拥有智能的敌人可以寻找、追逐并攻击玩家。在某些情况下，可能需要对敌人FOV进行调整和改进，以便更好地匹配游戏环境和角色类型，如图8.11所示。



图8.11 完成的关卡

## 8.6 小结

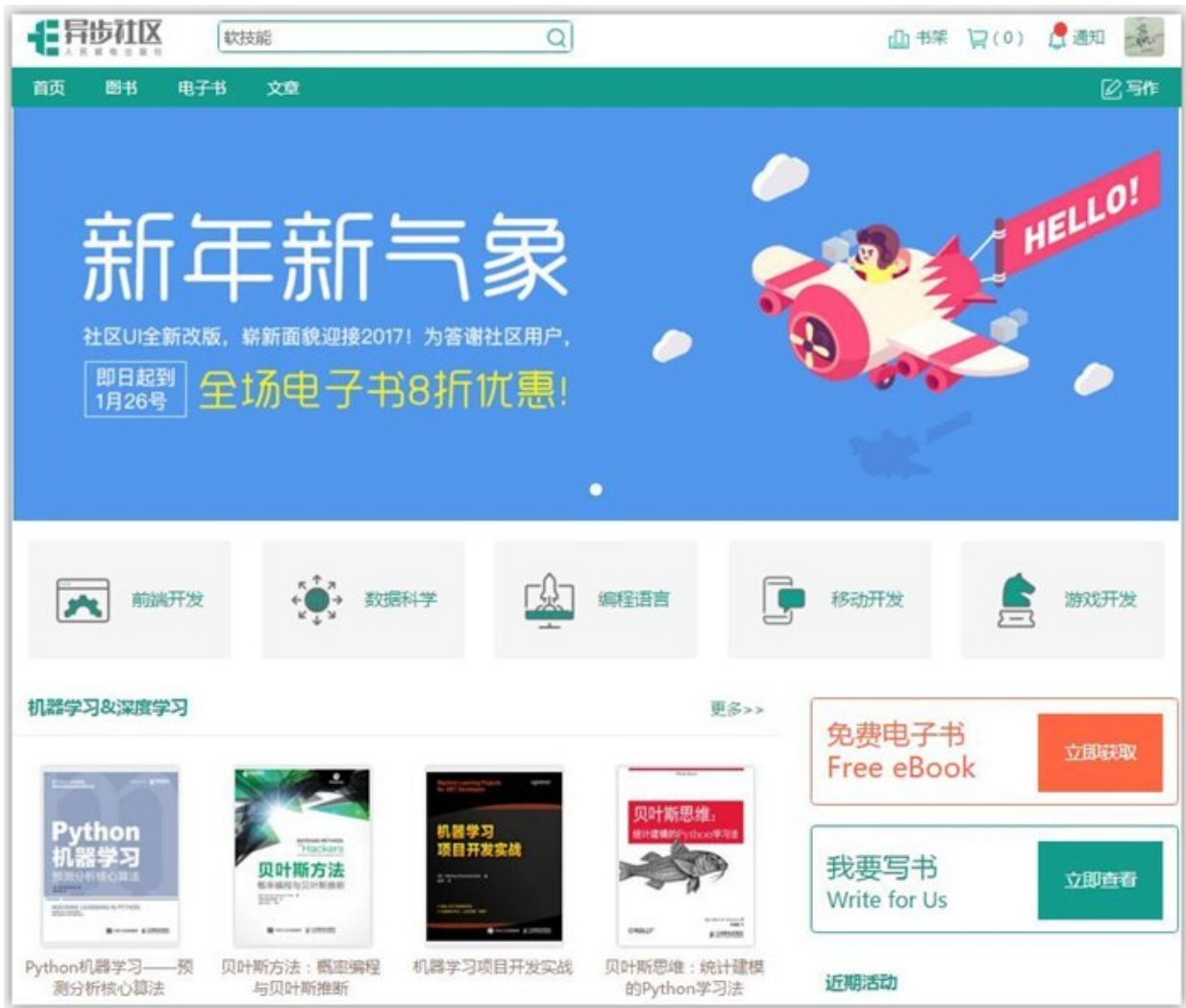
干得不错！现在到达了人工智能项目的结束部分，同时这也是本书的尾声了。在这个项目完成之时，已经创建了一个完整的地形、一个NPC预设体以及一系列协同工作实现了人工智能的脚本。对于游戏来说，人工智能指的不过是看起来有智能的对象以及创建它们的技术。至此，本书完成了4个项目，现在的你已经具备了多种开发二维和三维专业级游戏的关键技能，希望你再接再厉！

# 欢迎来到异步社区！

## 异步社区的来历

异步社区([www.epubit.com.cn](http://www.epubit.com.cn))是人民邮电出版社旗下IT专业图书旗舰社区，于2015年8月上线运营。

异步社区依托于人民邮电出版社20余年的IT专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与POD按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。



## 社区里都有什么？

### 购买图书

我们出版的图书涵盖主流IT技术，在编程语言、Web技术、数据科学等领域有众多经典畅销图书。社区现已上线图书1000余种，电子书400多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

### 下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

## 与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

## 灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。100积分=1元，购买图书时，在  里填入可使用的积分数值，即可扣减相应金额。

### 特别优惠

购买本电子书的读者专享**异步社区优惠券**。使用方法：注册成为社区用户，在下单购书时输入“**57AWG**”，然后点击“使用优惠码”，即可享受电子书8折优惠（本优惠券只可使用一次）。



## 纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。

The screenshot displays the book page for "Wireshark网络分析的艺术" (The Art of Network Analysis with Wireshark). The page includes the book cover, author information (林沛满), and a detailed description. It offers three purchase options: a discounted paper edition (¥31.50), an electronic edition (¥25.00), and a combined bundle (¥45.00). A "一起购买" (Buy Together) button is available for a total price of ¥75.60. The page also features a "本书作者" (Book Author) section with a profile for LinPeiman, a "兑换样书" (Exchange Sample Book) section with "立即兑换" (Exchange Now) and "如何赚取积分" (How to Earn Points) buttons, and an "电子书版本" (E-book Version) section with PDF, Epub, and Mobi options. A "精彩推荐" (Recommended) section at the bottom right suggests "Nmap渗透测试指南" (Nmap Penetration Testing Guide).

## 社区里还可以做什么？

### 提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得100积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

### 写作



社区提供基于Markdown的写作环境，喜欢写作的您可以在此一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

## 会议活动早知道

您可以掌握IT圈的技术会议资讯，更有机会免费获赠大会门票。

## 加入异步

扫描任意二维码都能找到我们：



异步社区



微信订阅号



微信服务号



官方微博



QQ群: 436746675

社区网址: [www.epubit.com.cn](http://www.epubit.com.cn)

官方微信: 异步社区

**官方微博：** @人邮异步社区， @人民邮电出版社-信息技术分社

**投稿&咨询：** [contact@epubit.com.cn](mailto:contact@epubit.com.cn)



# Unity Shader

## 入门精要

冯乐乐 著



中国工信出版集团



人民邮电出版社  
POSTS & TELECOM PRESS

# 目 录

[版权信息](#)

[内容提要](#)

[前言](#)

[第1篇 基础篇](#)

[第1章 欢迎来到Shader的世界](#)

[1.1 程序员的三大浪漫](#)

[1.2 本书结构](#)

[第2章 渲染流水线](#)

[2.1 综述](#)

[2.1.1 什么是流水线](#)

[2.1.2 什么是渲染流水线](#)

[2.2 CPU和GPU之间的通信](#)

[2.2.1 把数据加载到显存中](#)

[2.2.2 设置渲染状态](#)

[2.2.3 调用Draw Call](#)

[2.3 GPU流水线](#)

[2.3.1 概述](#)

[2.3.2 顶点着色器](#)

[2.3.3 裁剪](#)

[2.3.4 屏幕映射](#)

[2.3.5 三角形设置](#)

[2.3.6 三角形遍历](#)

[2.3.7 片元着色器](#)

[2.3.8 逐片元操作](#)

[2.3.9 总结](#)

[2.4 一些容易困惑的地方](#)

[2.4.1 什么是OpenGL/DirectX](#)



[2.4.2 什么是HLSL、GLSL、Cg](#)  
[2.4.3 什么是Draw Call](#)  
[2.4.4 什么是固定管线渲染](#)  
[2.5 那么，你明白什么是Shader了吗](#)  
[2.6 扩展阅读](#)

## [第3章 Unity Shader基础](#)

[3.1 Unity Shader概述](#)  
[3.1.1 一对好兄弟：材质和Unity Shader](#)  
[3.1.2 Unity中的材质](#)  
[3.1.3 Unity中的Shader](#)  
[3.2 Unity Shader的基础：ShaderLab](#)  
[什么是ShaderLab?](#)  
[3.3 Unity Shader的结构](#)  
[3.3.1 给我们的Shader起个名字](#)  
[3.3.2 材质和Unity Shader的桥梁：Properties](#)  
[3.3.3 重量级成员：SubShader](#)  
[3.3.4 留一条后路：Fallback](#)  
[3.3.5 ShaderLab还有其他的语义吗](#)  
[3.4 Unity Shader的形式](#)  
[3.4.1 Unity的宠儿：表面着色器](#)  
[3.4.2 最聪明的孩子：顶点/片元着色器](#)  
[3.4.3 被抛弃的角落：固定函数着色器](#)  
[3.4.4 选择哪种Unity Shader形式](#)  
[3.5 本书使用的Unity Shader形式](#)  
[3.6 答疑解惑](#)  
[3.6.1 Unity Shader != 真正的Shader](#)  
[3.6.2 Unity Shader和Cg/HLSL之间的关系](#)  
[3.6.3 我可以GLSL来写吗](#)  
[3.7 扩展阅读](#)

## [第4章 学习Shader所需的数学基础](#)

[4.1 背景：农场游戏](#)  
[4.2 笛卡儿坐标系](#)  
[4.2.1 二维笛卡儿坐标系](#)  
[4.2.2 三维笛卡儿坐标系](#)  
[4.2.3 左手坐标系和右手坐标系](#)

#### [4.2.4 Unity使用的坐标系](#)

#### [4.2.5 练习题](#)

### [4.3 点和矢量](#)

#### [4.3.1 点和矢量的区别](#)

#### [4.3.2 矢量运算](#)

#### [4.3.3 练习题](#)

### [4.4 矩阵](#)

#### [4.4.1 矩阵的定义](#)

#### [4.4.2 和矢量联系起来](#)

#### [4.4.3 矩阵运算](#)

#### [4.4.4 特殊的矩阵](#)

#### [4.4.5 行矩阵还是列矩阵](#)

#### [4.4.6 练习题](#)

### [4.5 矩阵的几何意义：变换](#)

#### [4.5.1 什么是变换](#)

#### [4.5.2 齐次坐标](#)

#### [4.5.3 分解基础变换矩阵](#)

#### [4.5.4 平移矩阵](#)

#### [4.5.5 缩放矩阵](#)

#### [4.5.6 旋转矩阵](#)

#### [4.5.7 复合变换](#)

### [4.6 坐标空间](#)

#### [4.6.1 为什么要使用这么多不同的坐标空间](#)

#### [4.6.2 坐标空间的变换](#)

#### [4.6.3 顶点的坐标空间变换过程](#)

#### [4.6.4 模型空间](#)

#### [4.6.5 世界空间](#)

#### [4.6.6 观察空间](#)

#### [4.6.7 裁剪空间](#)

#### [4.6.8 屏幕空间](#)

#### [4.6.9 总结](#)

### [4.7 法线变换](#)

## [4.8 Unity Shader的内置变量（数学篇）](#)

#### [4.8.1 变换矩阵](#)

#### [4.8.2 摄像机和屏幕参数](#)

### [4.9 答疑解惑](#)

#### [4.9.1 使用3×3还是4×4的变换矩阵](#)

[4.9.2 Cg中的矢量和矩阵类型](#)

[4.9.3 Unity中的屏幕坐标: ComputeScreenPos/VPOS/WPOS](#)

[4.10 扩展阅读](#)

[4.11 练习题答案](#)

## [第2篇 初级篇](#)

### [第5章 开始Unity Shader学习之旅](#)

[5.1 本书使用的软件和环境](#)

[5.2 一个最简单的顶点/片元着色器](#)

[5.2.1 顶点/片元着色器的基本结构](#)

[5.2.2 模型数据从哪里来](#)

[5.2.3 顶点着色器和片元着色器之间如何通信](#)

[5.2.4 如何使用属性](#)

[5.3 强大的援手: Unity提供的内置文件和变量](#)

[5.3.1 内置的包含文件](#)

[5.3.2 内置的变量](#)

[5.4 Unity提供的Cg/HLSL语义](#)

[5.4.1 什么是语义](#)

[5.4.2 Unity支持的语义](#)

[5.4.3 如何定义复杂的变量类型](#)

[5.5 程序员的烦恼: Debug](#)

[5.5.1 使用假彩色图像](#)

[5.5.2 利用神器: Visual Studio](#)

[5.5.3 最新利器: 帧调试器](#)

[5.6 小心: 渲染平台的差异](#)

[5.6.1 渲染纹理的坐标差异](#)

[5.6.2 Shader的语法差异](#)

[5.6.3 Shader的语义差异](#)

[5.6.4 其他平台差异](#)

[5.7 Shader整洁之道](#)

[5.7.1 float、half还是fixed](#)

[5.7.2 规范语法](#)

[5.7.3 避免不必要的计算](#)

[5.7.4 慎用分支和循环语句](#)

[5.7.5 不要除以0](#)

[5.8 扩展阅读](#)

## [第6章 Unity中的基础光照](#)

### [6.1 我们是如何看到这个世界的](#)

#### [6.1.1 光源](#)

#### [6.1.2 吸收和散射](#)

#### [6.1.3 着色](#)

#### [6.1.4 BRDF光照模型](#)

### [6.2 标准光照模型](#)

#### [6.2.1 环境光](#)

#### [6.2.2 自发光](#)

#### [6.2.3 漫反射](#)

#### [6.2.4 高光反射](#)

#### [6.2.5 逐像素还是逐顶点](#)

#### [6.2.6 总结](#)

### [6.3 Unity中的环境光和自发光](#)

### [6.4 在Unity Shader中实现漫反射光照模型](#)

#### [6.4.1 实践：逐顶点光照](#)

#### [6.4.2 实践：逐像素光照](#)

#### [6.4.3 半兰伯特模型](#)

### [6.5 在Unity Shader中实现高光反射光照模型](#)

#### [6.5.1 实践：逐顶点光照](#)

#### [6.5.2 实践：逐像素光照](#)

#### [6.5.3 Blinn-Phong光照模型](#)

### [6.6 召唤神龙：使用Unity内置的函数](#)

## [第7章 基础纹理](#)

### [7.1 单张纹理](#)

#### [7.1.1 实践](#)

#### [7.1.2 纹理的属性](#)

### [7.2 凹凸映射](#)

#### [7.2.1 高度纹理](#)

#### [7.2.2 法线纹理](#)

#### [7.2.3 实践](#)

#### [7.2.4 Unity中的法线纹理类型](#)

### [7.3 渐变纹理](#)

### [7.4 遮罩纹理](#)

#### [7.4.1 实践](#)

#### [7.4.2 其他遮罩纹理](#)

## [第8章 透明效果](#)

### [8.1 为什么渲染顺序很重要](#)

### [8.2 Unity Shader的渲染顺序](#)

### [8.3 透明度测试](#)

### [8.4 透明度混合](#)

### [8.5 开启深度写入的半透明效果](#)

### [8.6 ShaderLab的混合命令](#)

#### [8.6.1 混合等式和参数](#)

#### [8.6.2 混合操作](#)

#### [8.6.3 常见的混合类型](#)

### [8.7 双面渲染的透明效果](#)

#### [8.7.1 透明度测试的双面渲染](#)

#### [8.7.2 透明度混合的双面渲染](#)

## [第3篇 中级篇](#)

## [第9章 更复杂的光照](#)

### [9.1 Unity的渲染路径](#)

#### [9.1.1 前向渲染路径](#)

#### [9.1.2 顶点照明渲染路径](#)

#### [9.1.3 延迟渲染路径](#)

#### [9.1.4 选择哪种渲染路径](#)

### [9.2 Unity的光源类型](#)

#### [9.2.1 光源类型有什么影响](#)

#### [9.2.2 在前向渲染中处理不同的光源类型](#)

### [9.3 Unity的光照衰减](#)

#### [9.3.1 用于光照衰减的纹理](#)

#### [9.3.2 使用数学公式计算衰减](#)

### [9.4 Unity的阴影](#)

#### [9.4.1 阴影是如何实现的](#)

#### [9.4.2 不透明物体的阴影](#)

#### [9.4.3 使用帧调试器查看阴影绘制过程](#)

#### [9.4.4 统一管理光照衰减和阴影](#)

#### [9.4.5 透明度物体的阴影](#)

### [9.5 本书使用的标准Unity Shader](#)

## [第10章 高级纹理](#)

- [10.1 立方体纹理](#)
  - [10.1.1 天空盒子](#)
  - [10.1.2 创建用于环境映射的立方体纹理](#)
  - [10.1.3 反射](#)
  - [10.1.4 折射](#)
  - [10.1.5 菲涅耳反射](#)
- [10.2 渲染纹理](#)
  - [10.2.1 镜子效果](#)
  - [10.2.2 玻璃效果](#)
  - [10.2.3 渲染纹理 vs. GrabPass](#)
- [10.3 程序纹理](#)
  - [10.3.1 在Unity中实现简单的程序纹理](#)
  - [10.3.2 Unity的程序材质](#)

- [第11章 让画面动起来](#)
  - [11.1 Unity Shader中的内置变量（时间篇）](#)
  - [11.2 纹理动画](#)
    - [11.2.1 序列帧动画](#)
    - [11.2.2 滚动的背景](#)
  - [11.3 顶点动画](#)
    - [11.3.1 流动的河流](#)
    - [11.3.2 广告牌](#)
    - [11.3.3 注意事项](#)

## [第4篇 高级篇](#)

- [第12章 屏幕后处理效果](#)
  - [12.1 建立一个基本的屏幕后处理脚本系统](#)
  - [12.2 调整屏幕的亮度、饱和度和对比度](#)
  - [12.3 边缘检测](#)
    - [12.3.1 什么是卷积](#)
    - [12.3.2 常见的边缘检测算子](#)
    - [12.3.3 实现](#)
  - [12.4 高斯模糊](#)
    - [12.4.1 高斯滤波](#)
    - [12.4.2 实现](#)
  - [12.5 Bloom效果](#)



[12.6 运动模糊](#)

[12.7 扩展阅读](#)

[第13章 使用深度和法线纹理](#)

[13.1 获取深度和法线纹理](#)

[13.1.1 背后的原理](#)

[13.1.2 如何获取](#)

[13.1.3 查看深度和法线纹理](#)

[13.2 再谈运动模糊](#)

[13.3 全局雾效](#)

[13.3.1 重建世界坐标](#)

[13.3.2 雾的计算](#)

[13.3.3 实现](#)

[13.4 再谈边缘检测](#)

[13.5 扩展阅读](#)

[第14章 非真实感渲染](#)

[14.1 卡通风格的渲染](#)

[14.1.1 渲染轮廓线](#)

[14.1.2 添加高光](#)

[14.1.3 实现](#)

[14.2 素描风格的渲染](#)

[14.3 扩展阅读](#)

[14.4 参考文献](#)

[第15章 使用噪声](#)

[15.1 消融效果](#)

[15.2 水波效果](#)

[5.3 再谈全局雾效](#)

[15.4 扩展阅读](#)

[15.5 参考文献](#)

[第16章 Unity中的渲染优化技术](#)

[16.1 移动平台的特点](#)

[16.2 影响性能的因素](#)

[16.3 Unity中的渲染分析工具](#)

[16.3.1 认识Unity 5的渲染统计窗口](#)

[16.3.2 性能分析器的渲染区域](#)

- [16.3.3 再谈帧调试器](#)
- [16.3.4 其他性能分析工具](#)
- [16.4 减少draw call数目](#)
  - [16.4.1 动态批处理](#)
  - [16.4.2 静态批处理](#)
  - [16.4.3 共享材质](#)
  - [16.4.4 批处理的注意事项](#)
- [16.5 减少需要处理的顶点数目](#)
  - [16.5.1 优化几何体](#)
  - [16.5.2 模型的LOD技术](#)
  - [16.5.3 遮挡剔除技术](#)
- [16.6 减少需要处理的片元数目](#)
  - [16.6.1 控制绘制顺序](#)
  - [16.6.2 时刻警惕透明物体](#)
  - [16.6.3 减少实时光照和阴影](#)
- [16.7 节省带宽](#)
  - [16.7.1 减少纹理大小](#)
  - [16.7.2 利用分辨率缩放](#)
- [16.8 减少计算复杂度](#)
  - [16.8.1 Shader的LOD技术](#)
  - [16.8.2 代码方面的优化](#)
  - [16.8.3 根据硬件条件进行缩放](#)
- [16.9 扩展阅读](#)

## [第5篇 扩展篇](#)

### [第17章 Unity的表面着色器探秘](#)

- [17.1 表面着色器的一个例子](#)
- [17.2 编译指令](#)
  - [17.2.1 表面函数](#)
  - [17.2.2 光照函数](#)
  - [17.2.3 其他可选参数](#)
- [17.3 两个结构体](#)
  - [17.3.1 数据来源: Input结构体](#)
  - [17.3.2 表面属性: SurfaceOutput结构体](#)
- [17.4 Unity背后做了什么](#)
- [17.5 表面着色器实例分析](#)

## [17.6 Surface Shader的缺点](#)

## [第18章 基于物理的渲染](#)

### [18.1 PBS的理论和数学基础](#)

#### [18.1.1 光是什么](#)

#### [18.1.2 双向反射分布函数 \(BRDF\)](#)

#### [18.1.3 漫反射项](#)

#### [18.1.4 高光反射项](#)

#### [18.1.5 Unity中的PBS实现](#)

### [18.2 Unity 5的Standard Shader](#)

#### [18.2.1 它们是如何实现的](#)

#### [18.2.2 如何使用Standard Shader](#)

### [18.3 一个更加复杂的例子](#)

#### [18.3.1 设置光照环境](#)

#### [18.3.2 放置反射探针](#)

#### [18.3.3 调整材质](#)

#### [18.3.4 线性空间](#)

### [18.4 答疑解惑](#)

#### [18.4.1 什么是全局光照](#)

#### [18.4.2 什么是伽马校正](#)

#### [18.4.3 什么是HDR](#)

#### [18.4.4 那么，PBS适合什么样的游戏](#)

### [18.5 扩展阅读](#)

### [18.6 参考文献](#)

## [第19章 Unity 5更新了什么](#)

### [19.1 场景“更亮了”](#)

### [19.2 表面着色器更容易“报错了”](#)

### [19.3 当家做主：自己控制非统一缩放的网格](#)

### [19.4 固定管线着色器逐渐退出舞台](#)

## [第20章 还有更多内容吗](#)

### [20.1 如果你想深入了解渲染的话](#)

### [20.2 世界那么大](#)

### [20.3 参考文献](#)

[欢迎来到异步社区！](#)

[返回总目录](#)

## 版权信息

书名：Unity Shader入门精要

ISBN：978-7-115-42305-4

本书由人民邮电出版社发行数字版。版权所有，侵权必究。

---

您购买的人民邮电出版社电子书仅供您个人使用，未经授权，不得以任何方式复制和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

---

• 著 冯乐乐

责任编辑 张 涛

• 人民邮电出版社出版发行 北京市丰台区成寿寺路11号

邮编 100164 电子邮件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

- 读者服务热线: (010)81055410

反盗版热线: (010)81055315



## 内容提要

本书不仅要教会读者如何使用Unity Shader，更重要的是要帮助读者学习Unity中的一些渲染机制以及如何使用Unity Shader实现各种自定义的渲染效果，希望这本书可以为读者打开一扇新的大门，让读者离制作心目中优秀游戏的心愿更近一步。

本书的主要内容为：第1章讲解了学习Unity Shader应该从哪里着手；第2章讲解了现代GPU是如何实现整个渲染流水线的，这对理解Shader的工作原理有着非常重要的作用；第3章讲解Unity Shader的实现原理和基本语法；第4章学习Shader所需的数学知识，帮助读者克服学习Unity Shader时遇到的数学障碍；第5章通过实现一个简单的顶点/片元着色器案例，讲解常用的辅助技巧等；第6章学习如何在Shader中实现基本的光照模型；第7章讲述了如何在Unity Shader中使用法线纹理、遮罩纹理等基础纹理；第8章学习如何实现透明度测试和透明度混合等透明效果；第9章讲解复杂的光照实现；第10章讲解在Unity Shader中使用立方体纹理、渲染纹理和程序纹理等高级纹理；第11章学习用Shader实现纹理动画、顶点动画等动态效果；第12章讲解了屏幕后处理效果的屏幕特效；第13章使用深度纹理和法线纹理实现更多屏幕特效；第14章讲解非真实感渲染的算法，如卡通渲染、素描风格的渲染等；第15章讲解噪声在游戏渲染中的应用；第16章介绍了常见的优化技巧；第17章介绍用表面着色器实现渲染；第18章讲解基于物理渲染的技术；第19章讲解在升级Unity 5时可能出现的问题，并给出

解决方法；第20章介绍许多非常有价值的学习资料，以帮助读者进行更深入的学习。

本书适合Unity初学者、游戏开发者、程序员，也可以作为大专院校相关专业师生的学习用书，以及培训学校的培训教材。

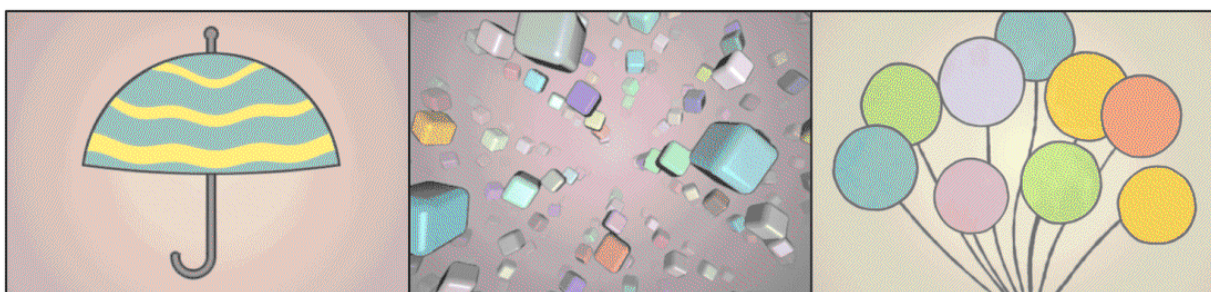
## 前言

2004年，有3位年轻人在开发他们的第一款游戏失利后，决定在丹麦首都哥本哈根建立一家游戏引擎公司。最初，他们的想法是要让全世界的开发人员可以使用最少的资源来创建出他们喜欢的游戏。谁也不曾想到，十年以后，这个起初并不起眼的公司已经发展成为游戏引擎公司巨头，而他们的游戏引擎也成为世界上应用最广泛的游戏引擎。没错，这个公司就是Unity Technologies，这3位年轻人分别是公司创始人David Helgason（CEO）、Nicholas Francis（CCO）和Joachim Ante（CTO）。而这3位创始人的初衷也得以实现，截止到2014年，全世界有超过300多万的开发者在使用游戏引擎Unity来开发游戏，更有6亿玩家在玩由Unity引擎制作的游戏。这股“Unity热”一直持续到现在。

虽然Unity引擎上手快，操作界面简单快捷，但许多Unity开发者却发现，当他们需要在Unity中实现一些特殊的画面效果时，往往无从下手。这些画面效果的实现通常和渲染有关，更具体来说，我们通常需要在Unity中编写一些Unity Shader文件来实现它们。一方面，对渲染知识的缺乏和对Shader的不了解导致很多开发者在这条路上举步维艰；另一方面，对游戏画面的提升是越来越多游戏公司的诉求。然而，Unity官方文档中不仅缺少对渲染原理讲解的内容，对Unity Shader本身的一些工作机制（概括来说，Unity Shader是Shader上层的一个抽象）同样缺少相关资料。同时，市面上能适应初学者的Unity Shader书少之又少，基于这些原因，使得我想要编写这样一本书来帮助开发者渡过困境。

本书旨在从基础开始，帮助读者逐渐了解并掌握如何编写Unity Shader。本书不仅仅是要教会读者“如何使用Unity Shader”，更重要的是要帮助读者建立对渲染流程的基本认识，在此基础上，帮助读者学习Unity中的一些渲染机制以及如何使用Unity Shader实现各种自定义的渲染效果。我相信，让读者首先了解原理再进行实践，相比于大量堆砌代码是更好的学习方法。因此，本书在开始实践前，均会为读者讲解大量的原理，让读者在学习时不再一头雾水。

尽管本书专注于学习Unity Shader，但根据我的学习经验来看，在不了解基础的渲染流程和基本的数学知识前，想要深入学习Shader的编写是非常困难的。实际上，Shader仅是整个渲染流程的一个子部分，因此，任何脱离渲染流程的对Shader的讲解可能会让读者更加困惑。而向量运算、矩阵变换等数学知识在Shader的编写中无处不在，因此，这些数学知识往往也是让初学者对Shader望而却步的原因。基于上面的两点观察，本书的安排从易到难，由基础到深入。我们把全书分为了5篇，读者可以在第1章中看到这些章节的具体安排。



随着硬件的发展，Shader的能力也越来越大。如果问你，一个Shader可以做什么？你可能会回答渲染游戏模型、模拟波动的海面、实现各种屏幕特效等。但如果告诉你，上面所示的3张图片完全依靠一个片元着色器来渲染实现，没有借助任何外部模型和纹理，你可能会觉

得非常不可思议！读者可以在Shadertoy网站上看到许多这样的例子。例如，上面的小雨伞、五彩的小方块，以及飘动的气球（由于本书是黑白印刷，一些效果无法显现）。一个简简单单的Shader可以做到什么程度的效果，我们已经不可预期。本书的重点不在于教导读者如何单纯使用Shader来实现上面的效果，而在于如何让Shader和其他游戏开发元素（例如，模型、纹理、脚本等）相配合，实现游戏中常见的渲染效果，我们在此只想说明Shader可能远比你想象的要强大得多。我们真诚地希望本书可以带领读者走进Shader的世界，让读者理解Shader、掌握Shader，和我们一起享受这样一个奇妙的游戏开发世界！

## 读这本书之前你需要哪些知识

本书面向Unity Shader初学者和程序员，尽量在本书的基础篇中介绍那些必要的基础知识，但仍然希望读者可以具备如下知识。

- 有一定（或少量）的编程经验。尽管Unity Shader的编写语言不同于C++、C#这种高级语言，但相比于完全没有编程经验的读者来说，学习过这些高级语言的读者更加容易理解Shader的代码。例如，什么是变量、什么是函数等。对于那些缺少编程经验但仍对Shader有浓厚兴趣的读者，一个好消息是，在Unity的帮助下，编写Unity Shader的代码量并不多，因此，这些读者仍然可以阅读本书。
- 对Unity引擎的操作界面比较熟悉。假定读者曾使用过一段时间的Unity，对其中的一些基本操作已经掌握。例如，如何创建场景、脚本和游戏对象等。

- 保持一定的耐心。我曾听到身边的一些朋友抱怨，为什么自己总是看不懂、学不会Shader，难道是自己学习能力有问题吗？实际上，这些朋友大多对Shader的学习缺乏耐心，总是抱着今天看一下明天就会的心情。但不幸的是，与C++、C#高级语言相比来说，就算我们成功编写了Shader版的“Hello world”，但对于为什么要这么写、它们是怎么执行的等一系列基础问题我们仍然并不理解。这正是我之前提到的，要想彻底理解Shader，就必须了解整个渲染流水线的工作方式。因此，保持耐心，打好基础，是每一个想要深入学习Shader的开发者的必经之路。
- 有一定的数学基础，包括了解基本的代数运算（如结合律、交换律等）、三角运算（如正弦、余弦计算等）。除此之外，如果读者具有大学水平的线性代数、微积分等数学知识，会发现阅读本书时会更加容易。为了帮助读者学习Shader中常见的数学运算，我们专门在本书的第4章为读者介绍向量、矩阵、空间变换等重要的数学内容。

如果你满足上面几点小小的条件，那么恭喜你，现在你可以安心地继续阅读本书了！

## 谁适合读这本书

任何想要了解渲染基础或想要自由地使用Unity Shader编写渲染效果的开发者均可阅读本书。这些开发者不仅限于进行游戏开发的程序员，也包括那些渴望更加自由地在Unity中实现各种画面效果的美工人员、在校学生和爱好者等。



## 为什么你需要这本书

与国内市场已有的介绍相关内容的书籍和资料相比来说，本书有一些独有的特色。

- 内容独特。本书填补了Unity Shader和渲染流水线之间的知识鸿沟，帮助读者打下良好的底层基础。同时，我们也会对Unity中一些渲染机制的工作原理进行详细剖析，帮助读者解决“是什么”“为什么”“怎么做”这3个基本问题。除此之外，本书配合大量实例，让读者在实践中逐渐掌握Unity Shader的编写。
- 结构连贯。由于网络上关于Unity Shader的资料非常零散，许多初学者总是无法系统地进行学习。本书在内容编排上颇费心思，从基础到进阶再到深入的讲解，解决读者长期以来的学习烦恼。
- 充分面向初学者。在本书的编写过程中，我一直在问自己，这么写到底读者能不能看懂？这使得在本书开头的几个章节中，尤其是在基础篇和初级篇中的章节中，我们的学习步调放得很慢，这是因为我非常了解在学习Shader的过程中哪些内容比较难理解，哪些内容非常容易让人困惑，而这些内容正是挡在初学者面前的拦路虎！为此，提供了大量的图示并配合文字说明，且在一些章节最后提供了“答疑解惑”小节来解释那些含糊不清而初学者又经常疑问的问题。考虑到数学往往是让初学者望而却步的重要因素，我们在第4章数学一章中特意安排了“农场游戏”这一背景案例，以这样一个虚拟的场景来帮助读者理解数学在渲染中是如何发挥作用的。

- 包含了Unity 5在渲染方面的新内容。例如，本书多次介绍Unity 5中的新工具帧调试器（**Frame Debugger**），并借助该工具的帮助来理解Unity中的渲染过程；第18章中介绍了Unity 5的基于物理的渲染（**PBR**），我们较为详细地剖析了PBR的实现原理，并介绍了如何在Unity 5中使用它们来实现一些更加真实的渲染效果。需要注意的是，在本书编写时使用的版本为当时的最新版本Unity 5.2.1（免费版），但本书出版时Unity可能会发布更新的版本，这可能会造成一些操作界面与本书内容有所冲突。例如，在Unity 5.3中，帧调试器的界面更加丰富，包含了材质属性等显示信息，但这并不影响阅读，我们在本书的勘误网址上会更新（[https://github.com/candycat1992/Unity\\_Shaders\\_Book](https://github.com/candycat1992/Unity_Shaders_Book)）。
- 补充了大量延伸阅读资料。渲染领域的博大精深绝不是一本书可以涵盖的，因此，在本书一些章节的最后，提供了“扩展阅读”小节，让那些希望更加深入学习的读者可以在提供的资料中找到更多的学习内容。

总而言之，我希望你可以从这本书中学到许多有价值的内容，并能够享受这个过程。相信我，这些内容很有趣。

## 本书源代码

读者可以在开源网站github（[https://github.com/candycat1992/Unity\\_Shaders\\_Book](https://github.com/candycat1992/Unity_Shaders_Book)）上下载本书的源代码。在编写本书时，我们使用的是当时Unity的最新版Unity 5.2.1（免费版），并在Mac 10.9.5平台和Windows 8平台下验证了代码的正确性。本书源代码的组

织方式大多按资源类型和章节进行划分，主要包含了以下关键文件夹。

文 件 夹	说 明
Assets/Scenes	包含了各章对应的场景，每个章节对应一个子文件夹，例如第7章所有场景所在的子文件夹为Assets/Scenes/Chapter7。每个场景的命名方式为Scene章号小节号_次小节号，例如7.2.3节对应的场景名为Scene_7_2_3。如果同一个小节包含了多个场景，那么会使用英文字母作为后缀依次表示，例如7.1.2节包含了两个场景Scene_7_1_2_a和Scene_7_1_2_b
Assets/Shaders	包含了各章实现的Unity Shader文件，每个章节对应一个子文件夹，例如第7章实现的所有Unity Shader所在的子文件夹为Assets/Shaders/Chapter7。每个Unity Shader的命名方式为ChapterX-功能，例如第7章使用渐变纹理的Unity Shader名为Chapter7-RampTexture
Assets/Materials	包含了各章对应的材质，每个章节对应一个子文件夹，例如第7章所有材质所在的子文件夹为Assets/ Scenes/Chapter7。每个材质的命名方式与它使用的Unity Shader名称相匹配，并以Mat作为后缀，例如使用名为Chapter7-RampTexture的Unity Shader的材质名称是RampTextureMat
Assets/Scripts	包含了各章对应的C#脚本，每个章节对应一个子文件夹，例如第5章所有脚本所在的子文件夹为Assets/Scripts/Chapter5
Assets/Textures	包含了各章使用的纹理贴图，每个章节对应一个子文件夹，例如第7章使用的所有纹理所在的子文件夹为Assets/Textures/Chapter7

除了上述文件夹外，源代码中还包含了一些辅助文件夹。例如，Assets/Editor文件夹中包含了一些需要在编辑器状态下运行的脚本，Assets/Prefabs文件夹下包含了各章使用的预设模型和其他常用预设模型等。

## 读者反馈

尽管我们在本书的编写过程中多次检查内容的正确性，但书中难免仍然会出现一些错误，欢迎读者批评指正。读者可以将问题反映到本书源代码所在的github讨论页（[https://github.com/candycat1992/Unity\\_Shaders\\_Book/issues](https://github.com/candycat1992/Unity_Shaders_Book/issues)），此网址也是本书源代码下载地址，该地址中也包括本书示例彩色图文档。也可以发邮件（[lelefeng1992@gmail.com](mailto:lelefeng1992@gmail.com)）联系作者，本书答疑QQ群为438103099。

编辑联系邮箱为[zhangtao@ptpress.com.cn](mailto:zhangtao@ptpress.com.cn)。

## 致谢

首先，我要感谢《Unity 3D ShaderLab开发实战详解》一书的作者郭浩瑜老师，是他向出版社的推荐才导致了本书的编写和出版，并给了我许多在书籍编写过程中的建议和帮助。

感谢卢鹏先生，在本书编写过程中，我们进行了很多关于优化、效果实现等方面的讨论，这些讨论让本书的内容更加丰富。卢先生的乐于分享和好学的精神让我十分敬佩。

我也要感谢我的家人，我的父母和姐姐，是你们在背后的默默支持让我走到了今天，永远爱你们。还要感谢我的男朋友之之，在我遇到瓶颈时，永远是你的鼓励和支持让我走出困境。也是你的帮助，让本书现在的封面得以呈现在读者面前。

除此之外，从开始编写本书到完成之时，很多网友给了我莫大的鼓励和可贵的建议，我从未想到有这么多素未谋面的朋友在关注着本书的进展，感谢你们，是你们让我更加有动力写完本书。

感谢宣雨松和罗盛誉老师在百忙之中为本书写推荐序，谢谢你们的鼓励和支持。

最后，我要感谢人民邮电出版社的编辑张涛，是您的热情鼓励让我对本书的未来满怀希望。感谢您对本书在内容编排、封面设计等方面的意见和建议，让这本书变得更好。感谢对本书进行修改和排版的出版社工作人员，是你们让这本书更完美地呈现在读者面前。

作者

## 第1篇 基础篇

这是很重要的一篇，尽管在本篇中我们没有进行真正的代码编写，但本篇会为初学者普及基本的理论知识以及必要的数学基础，为读者顺利步入Unity Shader学习打下很好的基础。

### 第1章 欢迎来到Shader的世界

欢迎来到Shader的世界！我们曾不断听到周围有人提出类似的问题：“Shader是什么”“我应该看哪些书才能学好Shader”“学习Unity Shader，我应该从哪里着手”。我希望这本书可以告诉你这些问题的答案。让你离制作心目中优秀游戏的心愿更近一步。

### 第2章 渲染流水线

这一章讲解了现代GPU是如何实现整个渲染流水线的，这些内容对于理解Shader的工作原理有着非常重要的作用。

### 第3章 Unity Shader基础

这一章将讲解Unity Shader的实现原理和基本语法，同时也将为读者解答一些常见的困惑点。

### 第4章 学习Shader所需的数学基础



数学向来是初学者面对的一大学习障碍。然而，在初级阶段的渲染学习中，我们需要掌握的数学理论实际上并不复杂。这一章将为读者讲解渲染过程中常见的数学知识。这章内容可以帮助读者理解 Shader 中的数学运算，我们在讲解过程中以一个具体的例子来阐述“一头奶牛的鼻子是如何一步步被绘制到屏幕上的”。

## 第1章 欢迎来到Shader的世界

欢迎来到Shader的世界！我们曾不断听到周围有人提出类似的问题：“Shader是什么”“我应该看哪些书才能学好Shader”“学习Unity Shader，我应该从哪里着手”。我们希望这本书可以告诉你这些问题的答案。如果本书是你学习Shader的第一本书，我们希望这本书可以为你打开一扇新的大门，让你离制作心目中的优秀游戏的心愿更近一步；如果不是，我们同样希望这本书可以让你更深入地理解Shader的方方面面，在学习Shader的过程中更上一层楼。

### 1.1 程序员的三大浪漫

有人说，程序员的三大浪漫是编译原理、操作系统和图形学（是的，我已经听到很多人在反驳这句话了，不要当真啦）。不管你是否认同这句话，我们只是想借此说明图形学在程序员心目中的地位。正在看此书的你，想必多多少少都对图形学或者渲染有一定兴趣，也许你想要通过此书来学习如何实现游戏中的各种特效，也许你仅仅是好奇那些绚丽的画面是如何产生的。我们是程序员中的“外貌协会”，期待着用代码编写出一个绚丽多姿的世界。这就是我们的浪漫。

我想，读者大概都经历过这样的场景：当你在游戏里看到那些出色的画面时，你很好奇这样的游戏是如何制作出来的，更具体的是，这样的渲染效果是如何得到的。于是你搜索后发现，这个游戏是Unity引擎开发的，更巧的是，Unity也是你熟知的引擎！于是你继续搜索，

想要知道如何在Unity里实现这样的效果，最后，你往往会得到“要编写自己的Shader”这样的答案。总算有了一些头绪，你继续在网上搜索如何学习编写Shader。于是你看到了很多文章，这些文章告诉你Unity Shader有哪些语法，一个普通的漫反射或者边缘高光的效果的代码是什么样子的。然后，你把这些代码粘贴到Unity中，保存后运行，效果出现了！一切看起来好像都很顺利，可是，当你仔细阅读这些代码时，却往往没有头绪。你不知道为什么要有一个名为vert和frag的函数，它们是什么时候调用的，为什么vert函数里要进行一些矩阵运算，这些矩阵是用来做什么的，为什么当你按照C#里面的一些语法编写时Shader却报错了。这些疑问大大影响了你学习Shader的信心，你开始觉得这是一个比学习C#难许多倍的事情，怀疑自己是不是还不具备学习如何编写Shader的基础。

如果上面的情景和你的经历有些类似，那么相信我，有很多人和你有一样的烦恼。事实上，我们之所以会觉得学习Shader比学习C#这样的编程语言更加困难，一个原因是因为Shader需要牵扯到整个渲染流程。当学习C++、C#这样的高级语言时，我们可以在不了解计算机架构的情况下仍然编写出实现各种功能的代码，这样的高级语言更符合人类的思维方式。然而，Shader并不是这样的。我们之所以要学习Shader，是想要学习如何把物体按照自己的意愿渲染到屏幕上，但是，Shader只是整个渲染流程中的一个子部分。虽然它很关键，但想要学习它，我们就需要了解整个渲染流程是如何进行的。和C++这样的高级语言不同，尽管Shader的编写语言已经达到了我们可以理解的程度，但Shader更多地是面向GPU的工作方式，所以它的一些语法对

我们来说并不那么直观。因此，任何一篇只讲语法、不讲渲染框架的文章都无法解决读者的困惑。

我们希望通过本书可以帮助读者建立一个渲染流程的整体体系，这些基础是跨越Shader学习中层层障碍的重要因素。我们也相信，在学习完本书后，读者可以自行回答本章开头提出的那些问题。

## 1.2 本书结构

我们在编写本书时尽量考虑到没有渲染基础的读者们。因此，我们把整书分成了五大篇。

- 基础篇

这是很重要的一篇，尽管在本篇中我们没有进行真正的代码编写，但基础篇会为初学者普及基本的理论知识以及必要的数学基础。基础篇包括了以下3个章节。

**第2章渲染流水线** 这一章讲解了现代GPU是如何实现整个渲染流水线的，这些内容对于理解Shader的工作原理有着非常重要的作用。

**第3章 Unity Shader基础** Unity在原有的渲染流程上进行了封装，并提供给开发者新的图像编程接口——Unity Shader。这一章将讲解Unity Shader的实现原理和基本语法，同时也将为读者解答一些常见的困惑点。

**第4章学习Shader所需的数学基础** 数学向来是初学者面对的一大学习障碍。然而，在初级阶段的渲染学习中，我们需要掌握的数学

理论实际并不复杂。本章将为读者讲解渲染过程中常见的数学知识，如矢量、矩阵运算、坐标空间等。本章内容可以大大帮助读者理解 Shader 中的数学运算。为了帮助读者加深理解，我们在讲解过程中以一个具体的例子来阐述“一头奶牛的鼻子是如何一步步被绘制到屏幕上的”。

- 初级篇

在学习完基础篇后，我们就正式开始了 Unity Shader 的学习之旅。初级篇将会从最简单的 Shader 开始，讲解 Shader 中基础的光照模型、纹理和透明效果等初级渲染效果。需要注意的是，我们在初级篇中实现的 Unity Shader 大多不能直接用于真实项目中，因为它们缺少了完整的光照计算，例如阴影、光照衰减等，仅仅是为了阐述一些实现原理。在第9章最后，我们会给出包括了完整光照计算的 Unity Shader。初级篇包含了以下4个章节。

**第5章开始 Unity Shader 学习之旅** 本章将实现一个简单的顶点/片元着色器，并详细解释其中每个步骤的原理，这需要读者对之前基础篇的内容有所理解。本章还会给出关于 Unity Shader 的一些常用的辅助技巧，例如如何调试、查看内置代码以及编写规范等。

**第6章 Unity 中的基础光照** 本章将学习如何在 Shader 中实现基本的光照模型，如漫反射、高光反射等。我们首先解释如何从无到有实现一个光照模型，最后给出使用 Unity 提供的内置函数来实现的版本。

**第7章基础纹理** 纹理的使用给渲染的世界带来了更多的变化。这一章将会讲述如何在Unity Shader中使用法线纹理、遮罩纹理等基础纹理。

**第8章透明效果** 透明是游戏中常用的渲染效果。这一章首先介绍了渲染的实现原理，并给出了和Unity的渲染顺序相关的重要内容。在了解了这些内容的基础上，我们将学习如何实现透明度测试和透明度混合等透明效果。

- 中级篇

中级篇是本书的进阶篇章，主要讲解Unity中的渲染路径、如何计算光照衰减和阴影、如何使用高级纹理和动画等一系列进阶内容。中级篇包含了以下3个章节。

**第9章更复杂的光照** 我们在初级篇中实现的光照模型没有考虑一些重要的光照计算，如阴影和光照衰减。本章首先讲解Unity中的3种渲染路径和3种重要的光源类型，再解释如何在前向渲染路径中实现包含了光照衰减、阴影等效果的完整的光照计算。在本章最后，我们会给出基于之前学习内容实现的包含了完整光照计算的Unity Shader。

**第10章高级纹理** 这一章将会讲解如何在Unity Shader中使用立方体纹理、渲染纹理和程序纹理等高级纹理。

**第11章让画面动起来** 静态的画面往往是无趣的。这一章将帮助读者学习如何在Shader中使用时间变量来实现纹理动画、顶点动画等动态效果。



- 高级篇

高级篇涵盖了一些Shader的高级用法，例如如何实现屏幕特效、利用法线和深度缓冲以及非真实感渲染等，同时，我们还会介绍一些针对移动平台的优化技巧。高级篇的结构如下。

**第12章屏幕后处理效果** 屏幕特效是游戏中常用的渲染手法之一。这一章将介绍如何在Unity中实现一个基本的屏幕后处理脚本系统，并给出一些基本的屏幕特效的实现原理，如高斯模糊、边缘检测等。

**第13章使用深度和法线纹理** 使用深度和法线纹理可以帮助我们实现很多屏幕特效。本章将介绍如何在Unity中获取这些特殊的纹理来实现屏幕特效。

**第14章非真实感渲染** 很多游戏使用了非真实感渲染的方法来渲染游戏画面。这一章将会给出常见的非真实感渲染的算法，如卡通渲染、素描风格的渲染等。本章的扩展阅读部分可以帮助读者找到更多其他类型的非真实感渲染的实现方法。

**第15章使用噪声** 很多时候噪声是我们的救星。本章给出了噪声在游戏渲染中的一些应用。

**第16章 Unity中的渲染优化技术** 优化往往是游戏渲染中的重点。这一章介绍了Unity中针对移动平台使用的常见的优化技巧。

- 扩展篇

扩展篇旨在进一步扩展读者的视野。本篇将会介绍Unity的表面着色器的实现机制，并介绍基于物理的渲染的相关内容。最后，我们给出了更多的关于学习渲染的资料。扩展篇包含了以下4个章节。

**第17章Unity的表面着色器探秘** Unity提出了一种新颖的Shader形式——表面着色器。本章将会介绍这些表面着色器是如何实现的，以及如何使用这些表面着色器来实现渲染。

**第18章基于物理的渲染** Unity 5终于引入了基于物理的渲染，这给Unity引擎带来了更强的渲染能力。这一章将介绍基于物理渲染的理论基础，并解释Unity是如何实现基于物理的渲染的。我们还会在本章实现一个基本的场景来进一步阐述如何在Unity 5中利用基于物理的渲染。

**第19章Unity 5更新了什么** 相较于Unity 4.x，Unity 5在Shader方面有很多重要的更新。本章将给出Unity 5中一些重要的更新，以帮助读者解决在升级Unity 5时所面对的各种问题。

**第20章还有更多内容吗** 图形学的丰富多彩远远超乎我们的想象，我们相信一本书也远远无法满足一些读者强烈的求知欲。在最后一章中，我们将给出许多非常有价值的学习资料，以帮助读者进行更深入的学习。

那么，你准备好了吗？和我们一起进入Shader的世界吧！

## 第2章 渲染流水线

在开始一切学习之前，我们有必要了解什么是**Shader**，即着色器。与之关系非常紧密的就是渲染流水线。可以说，如果你没有了解过渲染流水线的工作流程，就永远无法说自己对**Shader**已经入门。

渲染流水线的最终目的在于生成或者说是渲染一张二维纹理，即我们在电脑屏幕上看到的所有效果。它的输入是一个虚拟摄像机、一些光源、一些**Shader**以及纹理等。

本章将会给出渲染流水线的概览，同时会尽量避免数学上的计算，而仅仅提供一些全局上的描述。本书给出的流水线不仅适用于Unity平台，如果读者想要深入了解并学习着色器的话，会发现下面的内容同样是非常重要和有价值的。

### 2.1 综述

要学会怎么使用**Shader**，我们首先要了解**Shader**是怎么工作的。实际上，**Shader**仅仅是渲染流水线中的一个环节，想要让我们的**Shader**发挥出它的作用，我们就需要知道它在渲染流水线中扮演了怎样的角色。而本节会给出简化后的渲染流水线的工作流程。

#### 2.1.1 什么是流水线

我们先来看一下真实生活中的流水线是什么。在工业上，流水线被广泛应用在装配线上。

我们来举一个例子。假设，老王有一个生产洋娃娃的工厂，一个洋娃娃的生产流程可以分为4个步骤：第1步，制作洋娃娃的躯干；第2步，缝上眼睛和嘴巴；第3步，添加头发；第4步，给洋娃娃进行最后的产品包装。

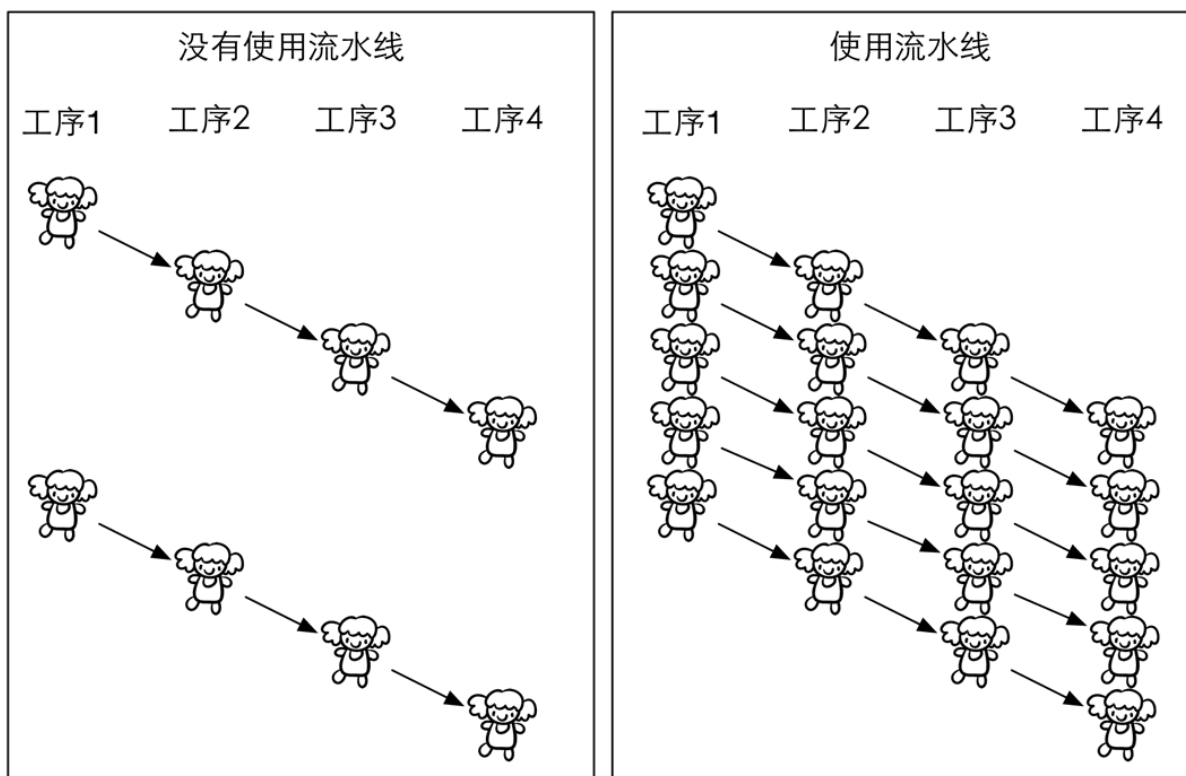
在流水线出现之前，只有在每个洋娃娃完成了所有这4个工序后才能开始制作下一个洋娃娃。如果说每个步骤需要的时间是1小时的话，那么每4个小时才能生产一个洋娃娃。

但后来人们发现了一个更加有效的方法，即使用流水线。老王把流水线引入工厂之后，工厂发生了很大的变化。虽然制作一个洋娃娃仍然需要4个步骤，但不需要从头到尾完成全部步骤，而是每个步骤由专人来完成，所有步骤并行进行。也就是说，当工序1完成了制作躯干的任务并把其交给工序2时，工序1又开始进行下一个洋娃娃的制作了。

使用流水线的好处在于可以提高单位时间的生产量。在洋娃娃的例子中，使用了流水线技术后每1个小时就可以生产一个洋娃娃。图2.1显示了使用流水线前后生产效率的变化。

可以发现，流水线系统中决定最后生产速度的是最慢的工序所需的时间。例如，如果生产洋娃娃的第二道工序需要的是两个小时，其他工序仍然需要1个小时的话，那么平均每两个小时才能生产出一个洋娃娃。即工序2是性能的瓶颈（bottleneck）。

理想情况下，如果把一个非流水线系统分成 $n$ 个流水线阶段，且每个阶段耗费时间相同的话，会使整个系统得到 $n$ 倍的速度提升。



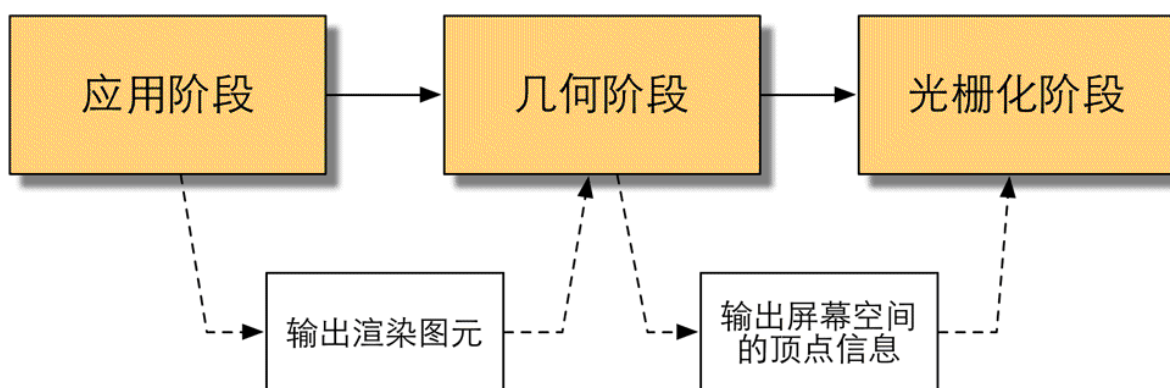
▲图2.1 真实生活中的流水线

### 2.1.2 什么是渲染流水线

上面的关于流水线的概念同样适用于计算机的图像渲染中。渲染流水线的工作任务在于由一个三维场景出发、生成（或者说渲染）一张二维图像。换句话说，计算机需要从一系列的顶点数据、纹理等信息出发，把这些信息最终转换成一张人眼可以看到的图像。而这个工作通常是由CPU和GPU共同完成的。

《Real-Time Rendering, Third Edition》<sup>[1]</sup>一书中将一个渲染流程分成3个阶段：应用阶段（Application Stage）、几何阶段（Geometry Stage）、光栅化阶段（Rasterizer Stage）。

注意，这里仅仅是概念性阶段，每个阶段本身通常也是一个流水线系统，即包含了子流水线阶段。图2.2显示了3个概念阶段之间的联系。



▲ 图2.2 渲染流水线中的3个概念阶段

- 应用阶段

从名字我们可以看出，这个阶段是由我们的应用主导的，因此通常由CPU负责实现。换句话说，我们这些开发者具有这个阶段的绝对控制权。

在这一阶段中，开发者有3个主要任务：首先，我们需要准备好场景数据，例如摄像机的位置、视锥体、场景中包含了哪些模型、使用了哪些光源等等；其次，为了提高渲染性能，我们往往需要做一个粗粒度剔除（culling）工作，以把那些不可见的物体剔除出去，这样就不



需要再移交给几何阶段进行处理；最后，我们需要设置好每个模型的渲染状态。这些渲染状态包括但不限于它使用的材质（漫反射颜色、高光反射颜色）、使用的纹理、使用的Shader等。这一阶段最重要的输出是渲染所需的几何信息，即**渲染图元（rendering primitives）**。通俗来讲，渲染图元可以是点、线、三角面等。这些渲染图元将会被传递给下一个阶段——几何阶段。

由于是由开发者主导这一阶段，因此应用阶段的流水线化是由开发者决定的。这不在本书的范畴内，有兴趣的读者可以参考本章的扩展阅读部分。

- 几何阶段

几何阶段用于处理所有和我们要绘制的几何相关的事情。例如，决定需要绘制的图元是什么，怎样绘制它们，在哪里绘制它们。这一阶段通常在GPU上进行。

几何阶段负责和每个渲染图元打交道，进行逐顶点、逐多边形的操作。这个阶段可以进一步分成更小的流水线阶段，这在下一章中会讲到。几何阶段的一个重要任务就是把顶点坐标变换到屏幕空间中，再交给光栅器进行处理。通过对输入的渲染图元进行多步处理后，这一阶段将会输出屏幕空间的二维顶点坐标、每个顶点对应的深度值、着色等相关信息，并传递给下一个阶段。

- 光栅化阶段

这一阶段将会使用上个阶段传递的数据来产生屏幕上的像素，并渲染出最终的图像。这一阶段也是在GPU上运行。光栅化的任务主要

是决定每个渲染图元中的哪些像素应该被绘制在屏幕上。它需要对上一个阶段得到的逐顶点数据（例如纹理坐标、顶点颜色等）进行插值，然后再进行逐像素处理。

和上一个阶段类似，光栅化阶段也可以分成更小的流水线阶段。



#### 提示

读者需要把上面的3个流水线阶段和我们将要讲到的GPU流水线阶段区分开来。这里的流水线均是概念流水线，是我们为了给一个渲染流程进行基本的功能划分而提出来的。下面要介绍的GPU流水线，则是硬件真正用于实现上述概念的流水线。

## 2.2 CPU和GPU之间的通信

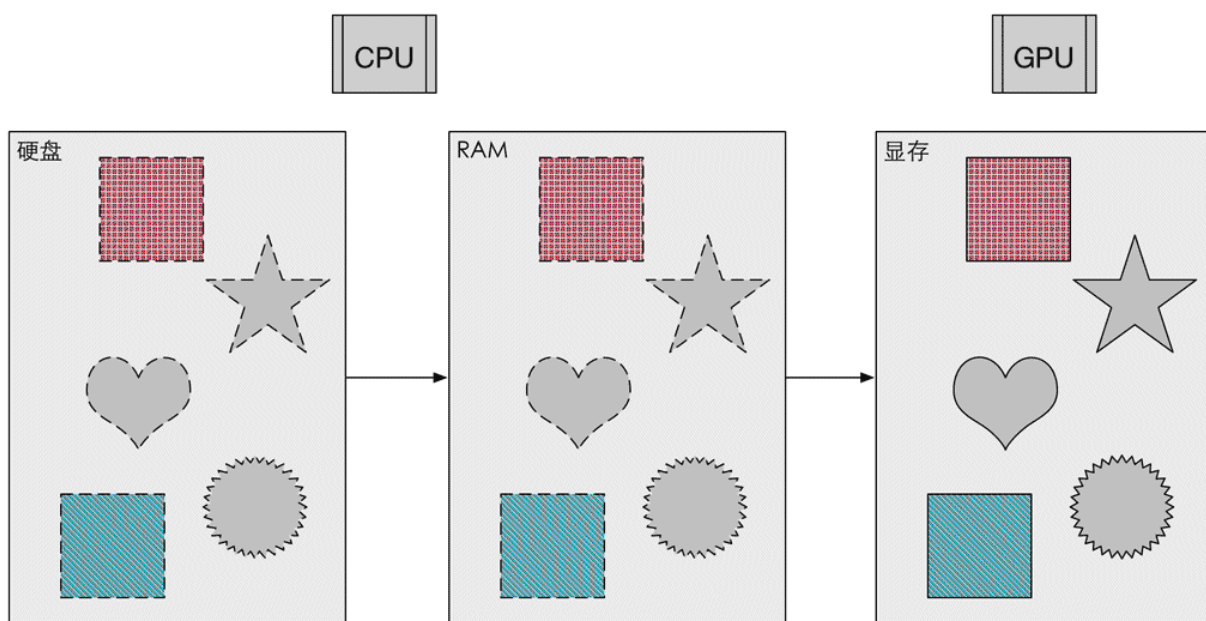
渲染流水线的起点是CPU，即应用阶段。应用阶段大致可分为下面3个阶段：

- （1）把数据加载到显存中。
- （2）设置渲染状态。
- （3）调用Draw Call（在本章的最后我们还会继续讨论它）。

### 2.2.1 把数据加载到显存中

所有渲染所需的数据都需要从硬盘（Hard Disk Drive，HDD）中加载到系统内存（Random Access Memory，RAM）中。然后，网格和纹

理等数据又被加载到显卡上的存储空间——显存（Video Random Access Memory, VRAM）中。这是因为，显卡对于显存的访问速度更快，而且大多数显卡对于RAM没有直接的访问权利。图2.3所示给出了这样一个例子。



▲图2.3 渲染所需的数据（两张纹理以及3个网格）从硬盘最终加载到显存中。在渲染时，GPU可以快速访问这些数据

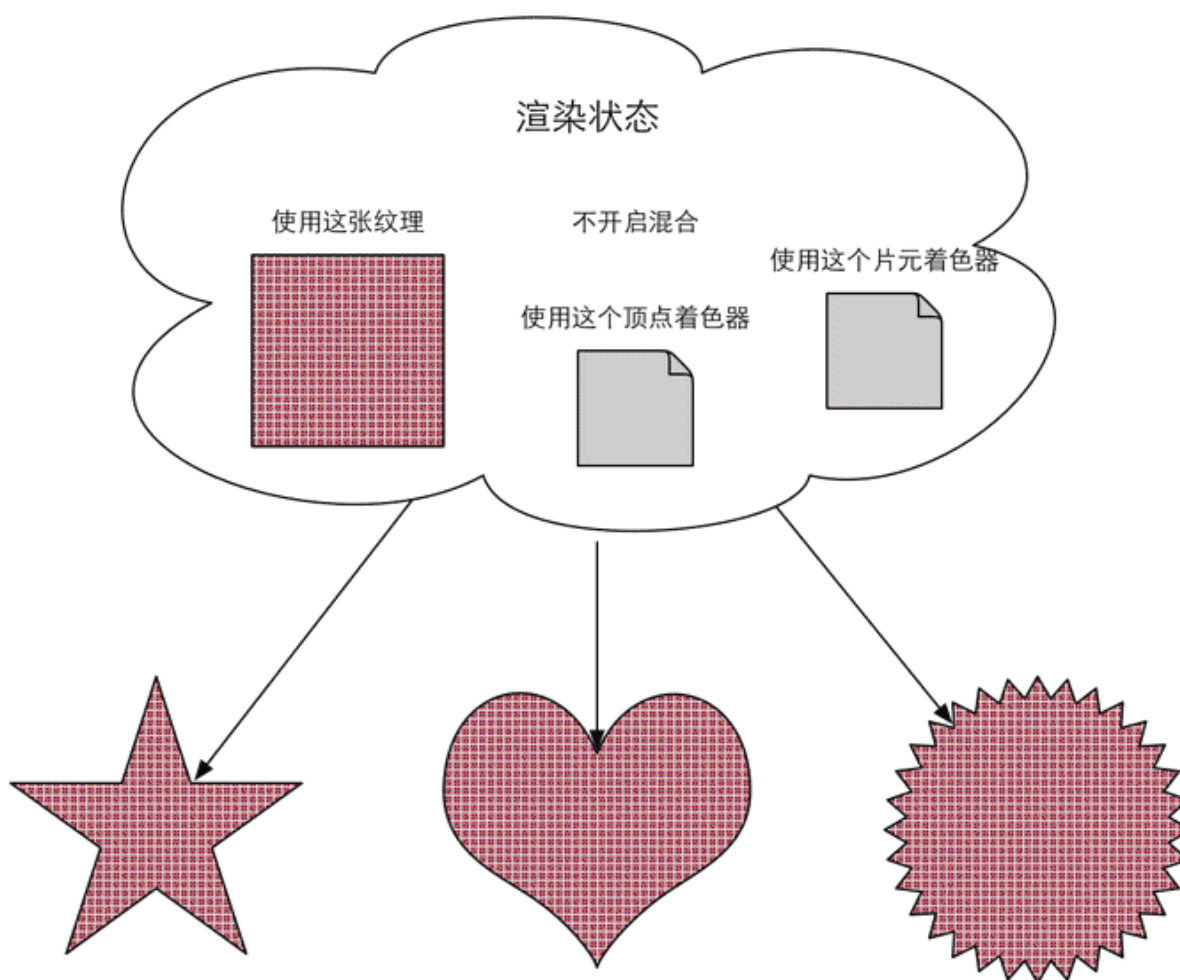
需要注意的是，真实渲染中需要加载到显存中的数据往往比图2.3所示复杂许多。例如，顶点的位置信息、法线方向、顶点颜色、纹理坐标等。

当把数据加载到显存中后，RAM中的数据就可以移除了。但对于一些数据来说，CPU仍然需要访问它们（例如，我们希望CPU可以访问网格数据来进行碰撞检测），那么我们可能就不希望这些数据被移除，因为从硬盘加载到RAM的过程是十分耗时的。

在这之后，开发者还需要通过CPU来设置渲染状态，从而“指导”GPU如何进行渲染工作。

### 2.2.2 设置渲染状态

什么是渲染状态呢？一个通俗的解释就是，这些状态定义了场景中的网格是怎样被渲染的。例如，使用哪个顶点着色器（Vertex Shader）/片元着色器（Fragment Shader）、光源属性、材质等。如果我们没有更改渲染状态，那么所有的网格都将使用同一种渲染状态。图2.4显示了当使用同一种渲染状态时，渲染3个不同网格的结果。



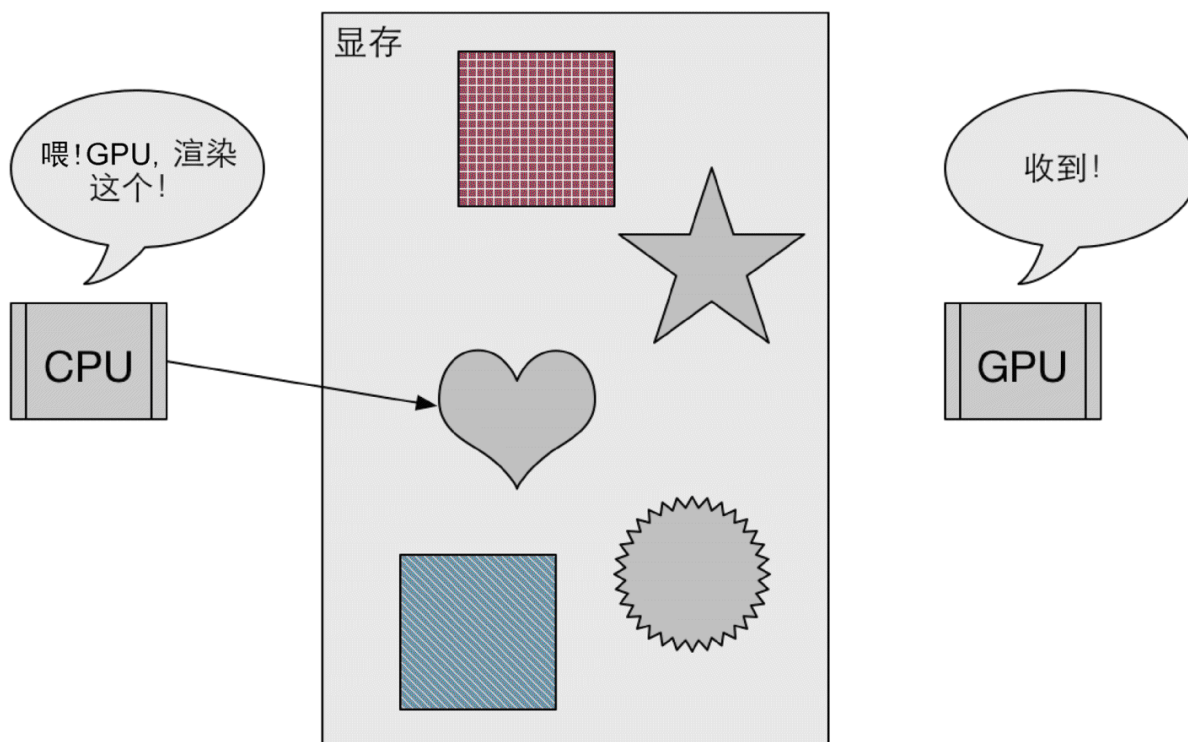
▲图2.4 在同一状态下渲染3个网格。由于没有更改渲染状态，因此3个网格的外观看起来像是同一种材质的物体

在准备好上述所有工作后，CPU就需要调用一个渲染命令来告诉GPU：“嘿！老兄，我都帮你把数据准备好啦，你可以按照我的设置来开始渲染啦！”而这个渲染命令就是Draw Call。

### 2.2.3 调用Draw Call

相信接触过渲染优化的读者应该都听说过Draw Call。实际上，Draw Call就是一个命令，它的发起方是CPU，接收方是GPU。这个命令仅仅会指向一个需要被渲染的图元（primitives）列表，而不会再包含任何材质信息——这是因为我们已经在上一个阶段中完成了！图2.5形象化地阐释了这个过程。

当给定了一个Draw Call时，GPU就会根据渲染状态（例如材质、纹理、着色器等）和所有输入的顶点数据来进行计算，最终输出成屏幕上显示的那些漂亮的像素。而这个计算过程，就是我们下一节要讲的GPU流水线。



▲ 图2.5 CPU通过调用Draw Call来告诉GPU开始进行一个渲染过程。一个Draw Call会指向本次调用需要渲染的图元列表

## 2.3 GPU流水线

当GPU从CPU那里得到渲染命令后，就会进行一系列流水线操作，最终把图元渲染到屏幕上。

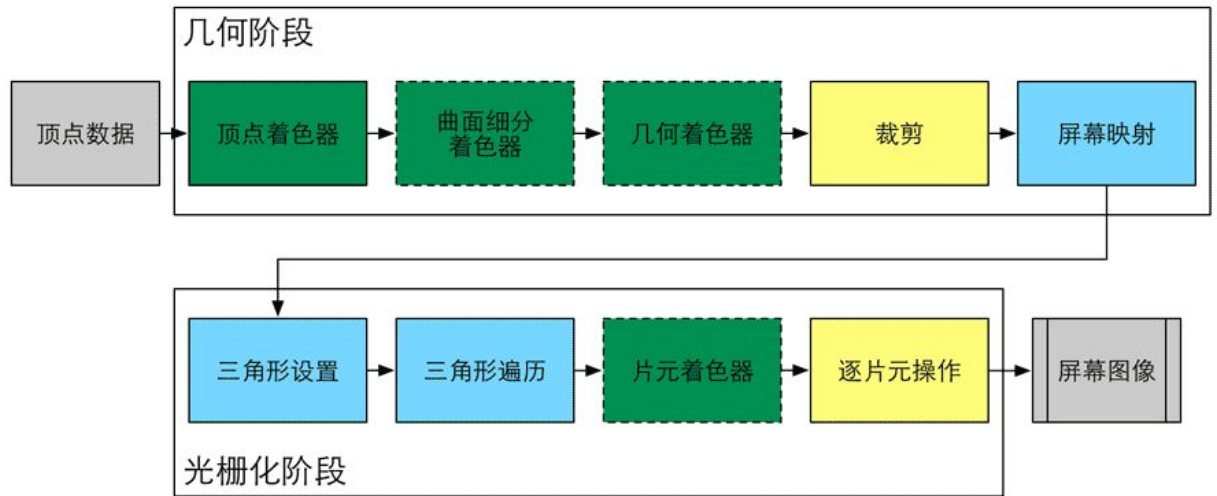
### 2.3.1 概述

在上一节中，我们解释了在应用阶段，CPU是如何和GPU通信，并通过调用Draw Call来命令GPU进行渲染。GPU渲染的过程就是GPU流水线。



对于概念阶段的后两个阶段，即几何阶段和光栅化阶段，开发者无法拥有绝对的控制权，其实现的载体是GPU。GPU通过实现流水线化，大大加快了渲染速度。虽然我们无法完全控制这两个阶段的实现细节，但GPU向开发者开放了很多控制权。在这一节中，我们将具体了解GPU是如何实现这两个概念阶段的。

几何阶段和光栅化阶段可以分成若干更小的流水线阶段，这些流水线阶段由GPU来实现，每个阶段GPU提供了不同的可配置性或可编程性。图2.6中展示了不同的流水线阶段以及它们的可配置性或可编程性。



▲ 图2.6 GPU的渲染流水线实现。颜色表示了不同阶段的可配置性或可编程性：绿色表示该流水线阶段是完全可编程控制的，黄色表示该流水线阶段可以配置但不是可编程的，蓝色表示该流水线阶段是由GPU固定实现的，开发者没有任何控制权。实线表示该Shader必须由开发者编程实现，虚线表示该Shader是可选的

从图中可以看出，GPU的渲染流水线接收顶点数据作为输入。这些顶点数据是由应用阶段加载到显存中，再由Draw Call指定的。这些数据随后被传递给顶点着色器。

**顶点着色器 (Vertex Shader)** 是完全可编程的，它通常用于实现顶点的空间变换、顶点着色等功能。**曲面细分着色器 (Tessellation Shader)** 是一个可选的着色器，它用于细分图元。**几何着色器**

**(Geometry Shader)** 同样是一个可选的着色器，它可以被用于执行逐图元 (Per-Primitive) 的着色操作，或者被用于产生更多的图元。下一个流水线阶段是**裁剪 (Clipping)**，这一阶段的目的是将那些不在摄像机视野内的顶点裁剪掉，并剔除某些三角图元的面片。这个阶段是可配置的。例如，我们可以使用自定义的裁剪平面来配置裁剪区域，也可以通过指令控制裁剪三角图元的正面还是背面。几何概念阶段的最后一个流水线阶段是**屏幕映射 (Screen Mapping)**。这一阶段是不可配置和编程的，它负责把每个图元的坐标转换到屏幕坐标系中。

光栅化概念阶段中的**三角形设置 (Triangle Setup)** 和**三角形遍历 (Triangle Traversal)** 阶段也都是固定函数 (Fixed-Function) 的阶段。接下来的**片元着色器 (Fragment Shader)**，则是完全可编程的，它用于实现逐片元 (Per-Fragment) 的着色操作。最后，**逐片元操作 (Per-Fragment Operations)** 阶段负责执行很多重要的操作，例如修改颜色、深度缓冲、进行混合等，它不是可编程的，但具有很高的可配置性。

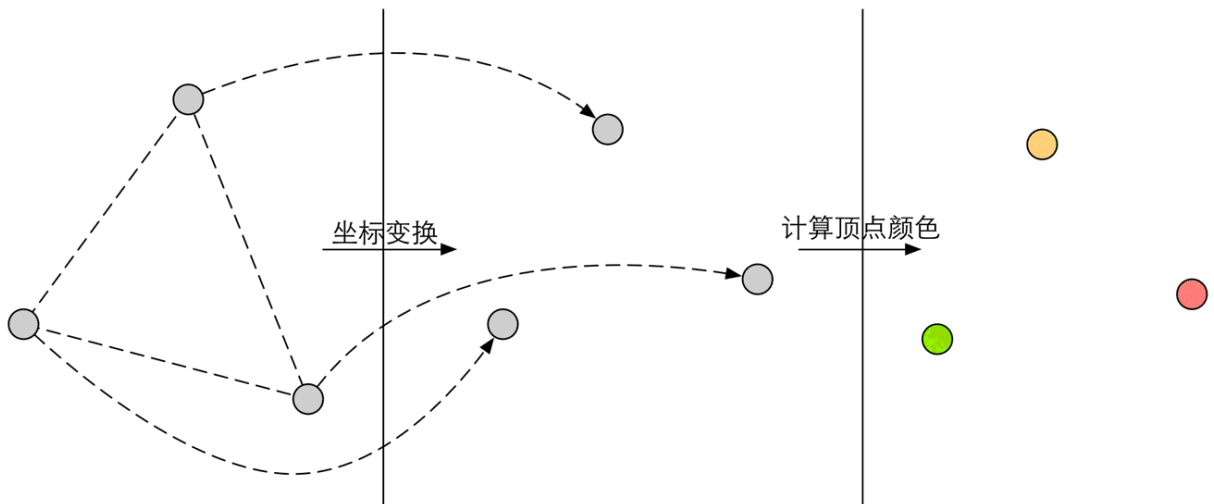
接下来，我们会对其中主要的流水线阶段进行更加详细的解释。

### 2.3.2 顶点着色器

**顶点着色器 (Vertex Shader)** 是流水线的第一个阶段，它的输入来自于CPU。顶点着色器的处理单位是顶点，也就是说，输入进来的每个顶点都会调用一次顶点着色器。顶点着色器本身不可以创建或者

销毁任何顶点，而且无法得到顶点与顶点之间的关系。例如，我们无法得知两个顶点是否属于同一个三角网格。但正是因为这样的相互独立性，GPU可以利用本身的特性并行化处理每一个顶点，这意味着这一阶段的处理速度会很快。

顶点着色器需要完成的工作主要有：坐标变换和逐顶点光照。当然，除了这两个主要任务外，顶点着色器还可以输出后续阶段所需的数据。图2.7展示了在顶点着色器中对顶点位置进行坐标变换并计算顶点颜色的过程。



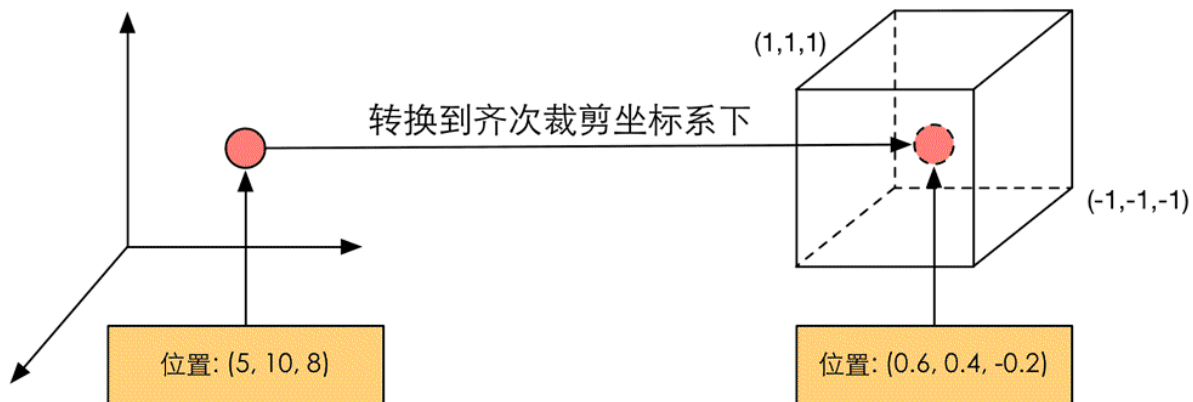
▲图2.7 GPU在每个输入的网格顶点上都会调用顶点着色器。顶点着色器必须进行顶点的坐标变换，需要时还可以计算和输出顶点的颜色。例如，我们可能需要进行逐顶点的光照

- 坐标变换。顾名思义，就是对顶点的坐标（即位置）进行某种变换。顶点着色器可以在这一步中改变顶点的位置，这在顶点动画中是非常有用的。例如，我们可以通过改变顶点位置来模拟水面、布料等。但需要注意的是，无论我们在顶点着色器中怎样改变顶点的位置，一个最基本的顶点着色器必须完成的一个工作

是，把顶点坐标从模型空间转换到齐次裁剪空间。想想看，我们在顶点着色器中是不是会看到类似下面的代码：

```
o.pos = mul(UNITY_MVP, v.position);
```

类似上面这句代码的功能，就是把顶点坐标转换到齐次裁剪坐标系下，接着通常再由硬件做透视除法后，最终得到归一化的设备坐标（Normalized Device Coordinates，NDC）。具体数学上的实现细节我们会在第4章中讲到。图2.8展示了这样的一个转换过程。



▲ 图2.8 顶点着色器会将模型顶点的位置变换到齐次裁剪坐标空间下，进行输出后再由硬件做透视除法得到NDC下的坐标

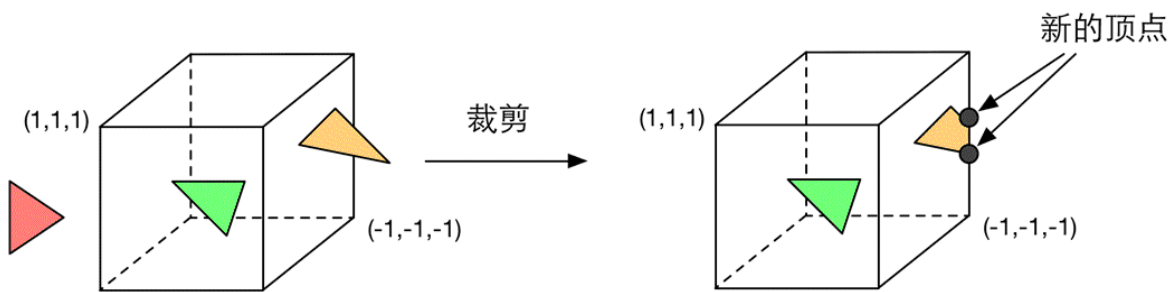
需要注意的是，图2.8给出的坐标范围是OpenGL同时也是Unity使用的NDC，它的z分量范围在 $[-1, 1]$ 之间，而在DirectX中，NDC的z分量范围是 $[0, 1]$ 。顶点着色器可以有不同的输出方式。最常见的输出路径是经光栅化后交给片元着色器进行处理。而在现代的Shader Model中，它还可以把数据发送给曲面细分着色器或几何着色器，感兴趣的读者可以自行了解。

### 2.3.3 裁剪

由于我们的场景可能会很大，而摄像机的视野范围很有可能不会覆盖所有的场景物体，一个很自然的想法就是，那些不在摄像机视野范围的物体不需要被处理。而**裁剪（Clipping）**就是为了完成这个目的而被提出来的。

一个图元和摄像机视野的关系有3种：完全在视野内、部分在视野内、完全在视野外。完全在视野内的图元就继续传递给下一个流水线阶段，完全在视野外的图元不会继续向下传递，因为它们不需要被渲染。而那些部分在视野内的图元需要进行一个处理，这就是裁剪。例如，一条线段的一个顶点在视野内，而另一个顶点不在视野内，那么在视野外部的顶点应该使用一个新的顶点来代替，这个新的顶点位于这条线段和视野边界的交点处。

由于我们已知在NDC下的顶点位置，即顶点位置在一个立方体内，因此裁剪就变得很简单：只需要将图元裁剪到单位立方体内。图2.9展示了这样的过程。



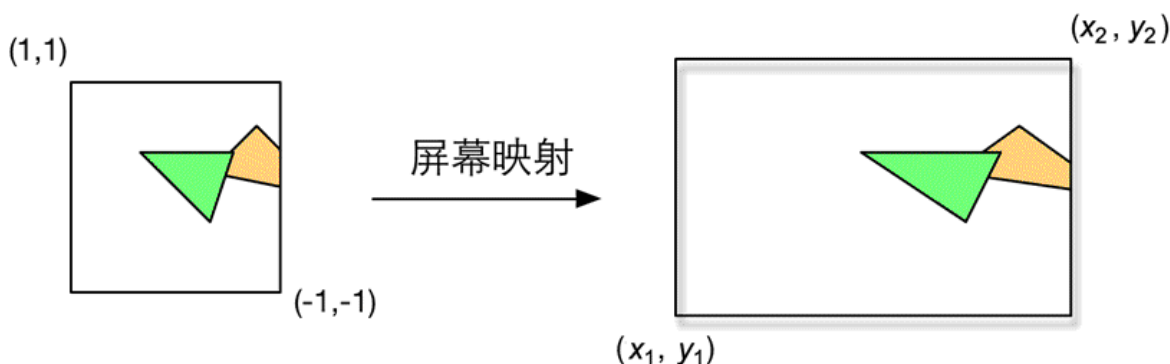
▲图2.9 只有在单位立方体的图元才需要被继续处理。因此，完全在单位立方体外部的图元（红色三角形）被舍弃，完全在单位立方体内部的图元（绿色三角形）将被保留。和单位立方体相交的图元（黄色三角形）会被裁剪，新的顶点会被生成，原来在外部的顶点会被舍弃

和顶点着色器不同，这一步是不可编程的，即我们无法通过编程来控制裁剪的过程，而是硬件上的固定操作，但我们可以自定义一个裁剪操作来对这一步进行配置。

### 2.3.4 屏幕映射

这一步输入的坐标仍然是三维坐标系下的坐标（范围在单位立方体内）。**屏幕映射（Screen Mapping）**的任务是把每个图元的 $x$ 和 $y$ 坐标转换到**屏幕坐标系（Screen Coordinates）**下。屏幕坐标系是一个二维坐标系，它和我们用于显示画面的分辨率有很大关系。

假设，我们需要把场景渲染到一个窗口上，窗口的范围是从最小的窗口坐标 $(x_1, y_1)$ 到最大的窗口坐标 $(x_2, y_2)$ ，其中 $x_1 < x_2$ 且 $y_1 < y_2$ 。由于我们输入的坐标范围在 $-1$ 到 $1$ ，因此可以想象到，这个过程实际是一个缩放的过程，如图2.10所示。你可能会问，那么输入的 $z$ 坐标会怎么样呢？屏幕映射不会对输入的 $z$ 坐标做任何处理。实际上，屏幕坐标系和 $z$ 坐标一起构成了一个坐标系，叫做**窗口坐标系（Window Coordinates）**。这些值会一起被传递到光栅化阶段。

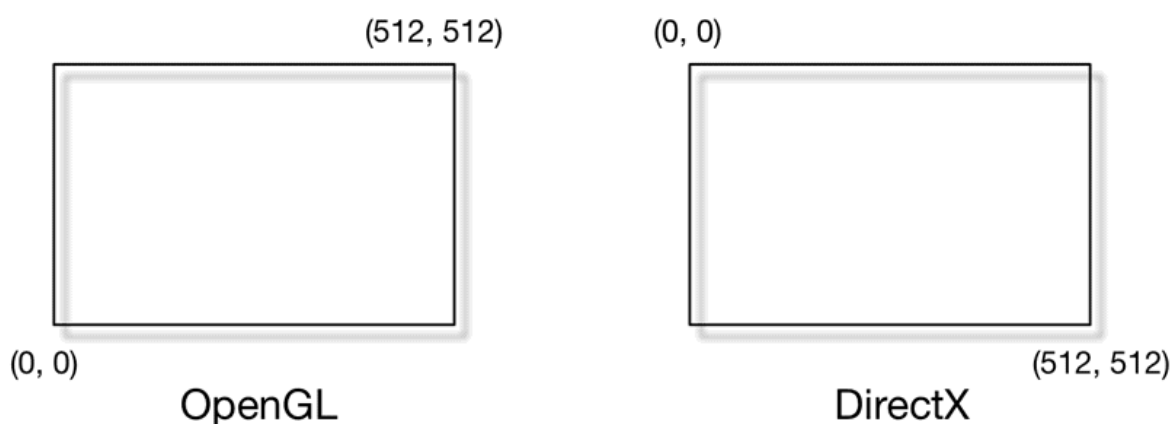


▲ 图2.10 屏幕映射将 $x$ 、 $y$ 坐标从 $(-1, 1)$ 范围转换到屏幕坐标系中



屏幕映射得到的屏幕坐标决定了这个顶点对应屏幕上哪个像素以及距离这个像素有多远。

有一个需要引起注意的地方是，屏幕坐标系在OpenGL和DirectX之间的差异问题。OpenGL把屏幕的左下角当成最小的窗口坐标值，而DirectX则定义了屏幕的左上角为最小的窗口坐标值。图2.11显示了这样的差异。



▲ 图2.11 OpenGL和DirectX的屏幕坐标系差异。对于一张512\*512大小的图像，在OpenGL中其(0, 0)点在左下角，而在DirectX中其(0, 0)点在左上角

产生这种差异的原因是，微软的窗口都使用了这样的坐标系，因为这和我们的阅读方式是一致的：从左到右、从上到下，并且很多图像文件也是按照这样的格式进行存储的。

不管原因如何，差异就这么造成了。留给我们开发者的就是，要时刻小心这样的差异，如果你发现得到的图像是倒转的，那么很有可能就是这个原因造成的。

### 2.3.5 三角形设置

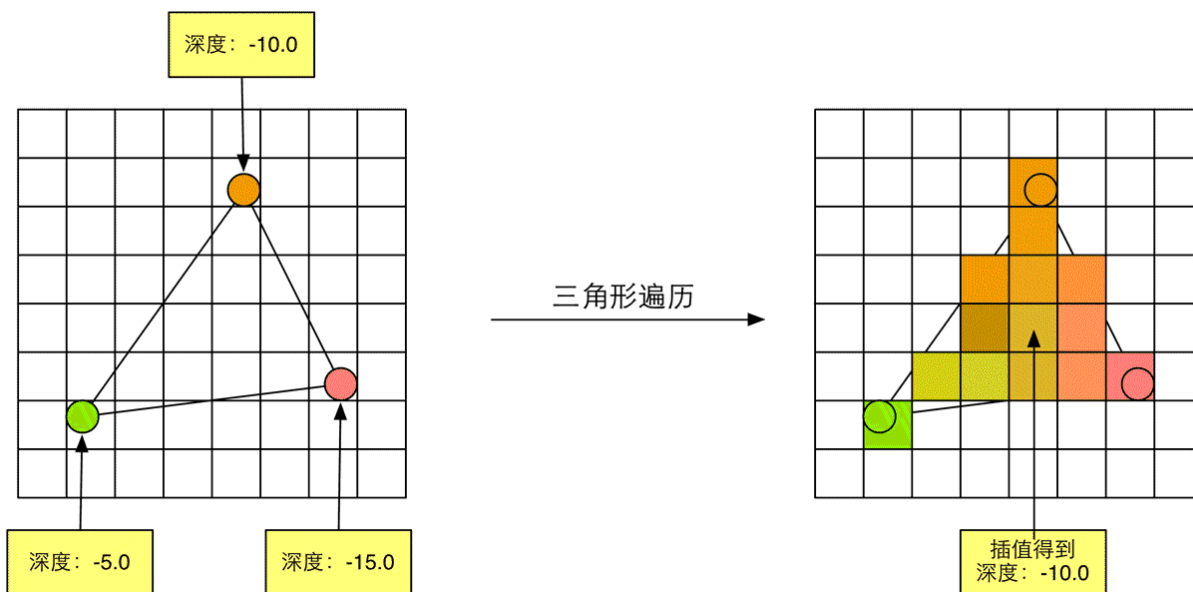
由这一步开始就进入了光栅化阶段。从上一个阶段输出的信息是屏幕坐标系下的顶点位置以及和它们相关的额外信息，如深度值（z坐标）、法线方向、视角方向等。光栅化阶段有两个最重要的目标：计算每个图元覆盖了哪些像素，以及为这些像素计算它们的颜色。

光栅化的第一个流水线阶段是**三角形设置（Triangle Setup）**。这个阶段会计算光栅化一个三角网格所需的信息。具体来说，上一个阶段输出的都是三角网格的顶点，即我们得到的是三角网格每条边的两个端点。但如果要得到整个三角网格对像素的覆盖情况，我们就必须计算每条边上的像素坐标。为了能够计算边界像素的坐标信息，我们就需要得到三角形边界的表示方式。这样一个计算三角网格表示数据的过程就叫做三角形设置。它的输出是为了给下一个阶段做准备。

### 2.3.6 三角形遍历

**三角形遍历（Triangle Traversal）**阶段将会检查每个像素是否被一个三角网格所覆盖。如果被覆盖的话，就会生成一个**片元（fragment）**。而这样一个找到哪些像素被三角网格覆盖的过程就是三角形遍历，这个阶段也被称为**扫描变换（Scan Conversion）**。

三角形遍历阶段会根据上一个阶段的计算结果来判断一个三角网格覆盖了哪些像素，并使用三角网格3个顶点的顶点信息对整个覆盖区域的像素进行插值。图2.12展示了三角形遍历阶段的简化计算过程。



▲图2.12 三角形遍历的过程。根据几何阶段输出的顶点信息，最终得到该三角网格覆盖的像素位置。对应像素会生成一个片元，而片元中的状态是对3个顶点的信息进行插值得到的。例如，对图2.12中3个顶点的深度进行插值得到其重心位置对应的片元的深度值为-10.0

这一步的输出就是得到一个片元序列。需要注意的是，一个片元并不是真正意义上的像素，而是包含了很多状态的集合，这些状态用于计算每个像素的最终颜色。这些状态包括了（但不限于）它的屏幕坐标、深度信息，以及其他从几何阶段输出的顶点信息，例如法线、纹理坐标等。

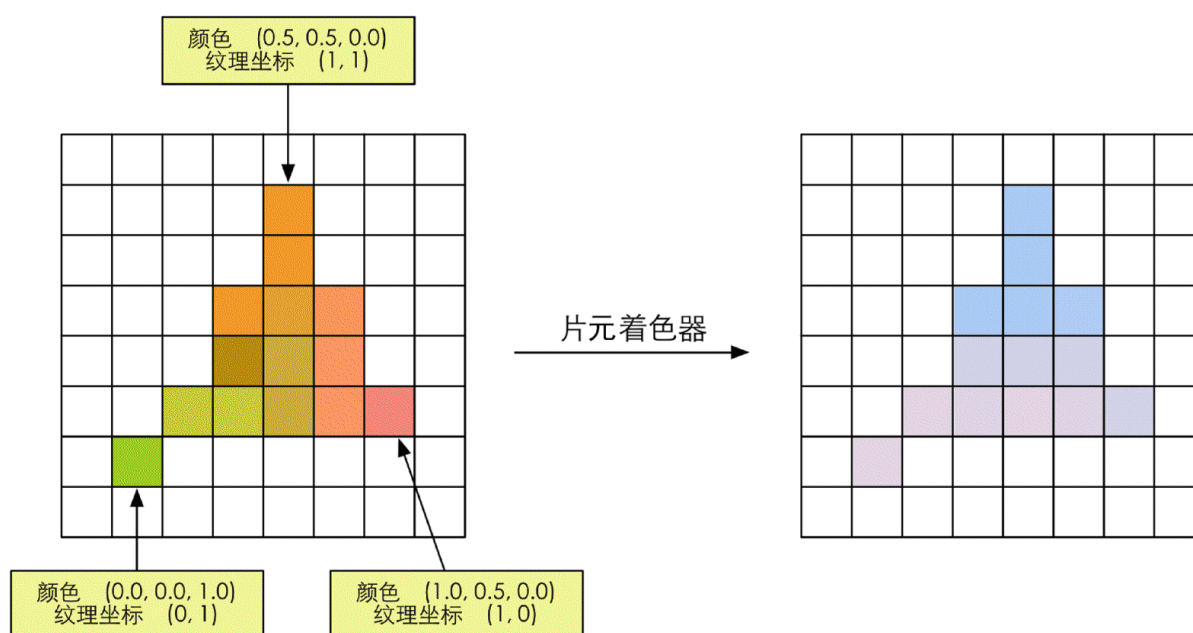
### 2.3.7 片元着色器

**片元着色器（Fragment Shader）**是另一个非常重要的可编程着色器阶段。在DirectX中，片元着色器被称为**像素着色器（Pixel Shader）**，但片元着色器是一个更合适的名字，因为此时的片元并不是一个真正意义上的像素。

前面的光栅化阶段实际上并不会影响屏幕上每个像素的颜色值，而是会产生一系列的数据信息，用来表述一个三角网格是怎样覆盖每个像素的。而每个片元就负责存储这样一系列数据。真正会对像素产生影响的阶段是下一个流水线阶段——**逐片元操作（Per-Fragment Operations）**。我们随后就会讲到。

片元着色器的输入是上一个阶段对顶点信息插值得到的结果，更具体来说，是根据那些从顶点着色器中输出的数据插值得到的。而它的输出是一个或者多个颜色值。图2.13显示了这样一个过程。

这一阶段可以完成很多重要的渲染技术，其中最重要的技术之一就是纹理采样。为了在片元着色器中进行纹理采样，我们通常会在顶点着色器阶段输出每个顶点对应的纹理坐标，然后经过光栅化阶段对三角网格的3个顶点对应的纹理坐标进行插值后，就可以得到其覆盖的片元的纹理坐标了。



▲图2.13 根据上一步插值后的片元信息，片元着色器计算该片元的输出颜色

虽然片元着色器可以完成很多重要效果，但它的局限在于，它仅可以影响单个片元。也就是说，当执行片元着色器时，它不可以将自己的任何结果直接发送给它的邻居们。有一个情况例外，就是片元着色器可以访问到导数信息（**gradient**，或者说是**derivative**）。有兴趣的读者可以参考本章的扩展阅读部分。

### 2.3.8 逐片元操作

终于到了渲染流水线的最后一步。**逐片元操作（Per-Fragment Operations）**是OpenGL中的说法，在DirectX中，这一阶段被称为**输出合并阶段（Output-Merger）**。**Merger**这个词可能更容易让读者明白这一步骤的目的：合并。而OpenGL中的名字可以让读者明白这个阶段的操作单位，即是对每一个片元进行一些操作。那么问题来了，要合并哪些数据？又要进行哪些操作呢？

这一阶段有几个主要任务。

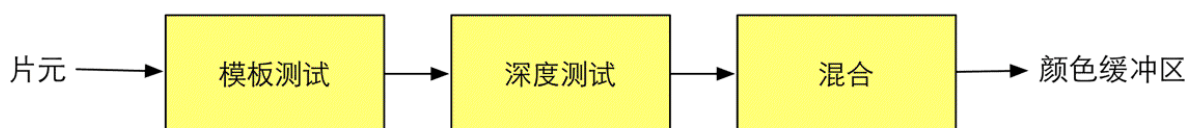
（1）决定每个片元的可见性。这涉及了很多测试工作，例如深度测试、模板测试等。

（2）如果一个片元通过了所有的测试，就需要把这个片元的颜色值和已经存储在颜色缓冲区中的颜色进行合并，或者说是混合。

需要指明的是，逐片元操作阶段是高度可配置性的，即我们可以设置每一步的操作细节。这在后面会讲到。

这个阶段首先需要解决每个片元的可见性问题。这需要进行一系列测试。这就好比考试，一个片元只有通过了所有的考试，才能最终获得和GPU谈判的资格，这个资格指的是它可以和颜色缓冲区进行合并。如果它没有通过其中的某一个测试，那么对不起，之前为了产生这个片元所做的所有工作都是白费的，因为这个片元会被舍弃掉。

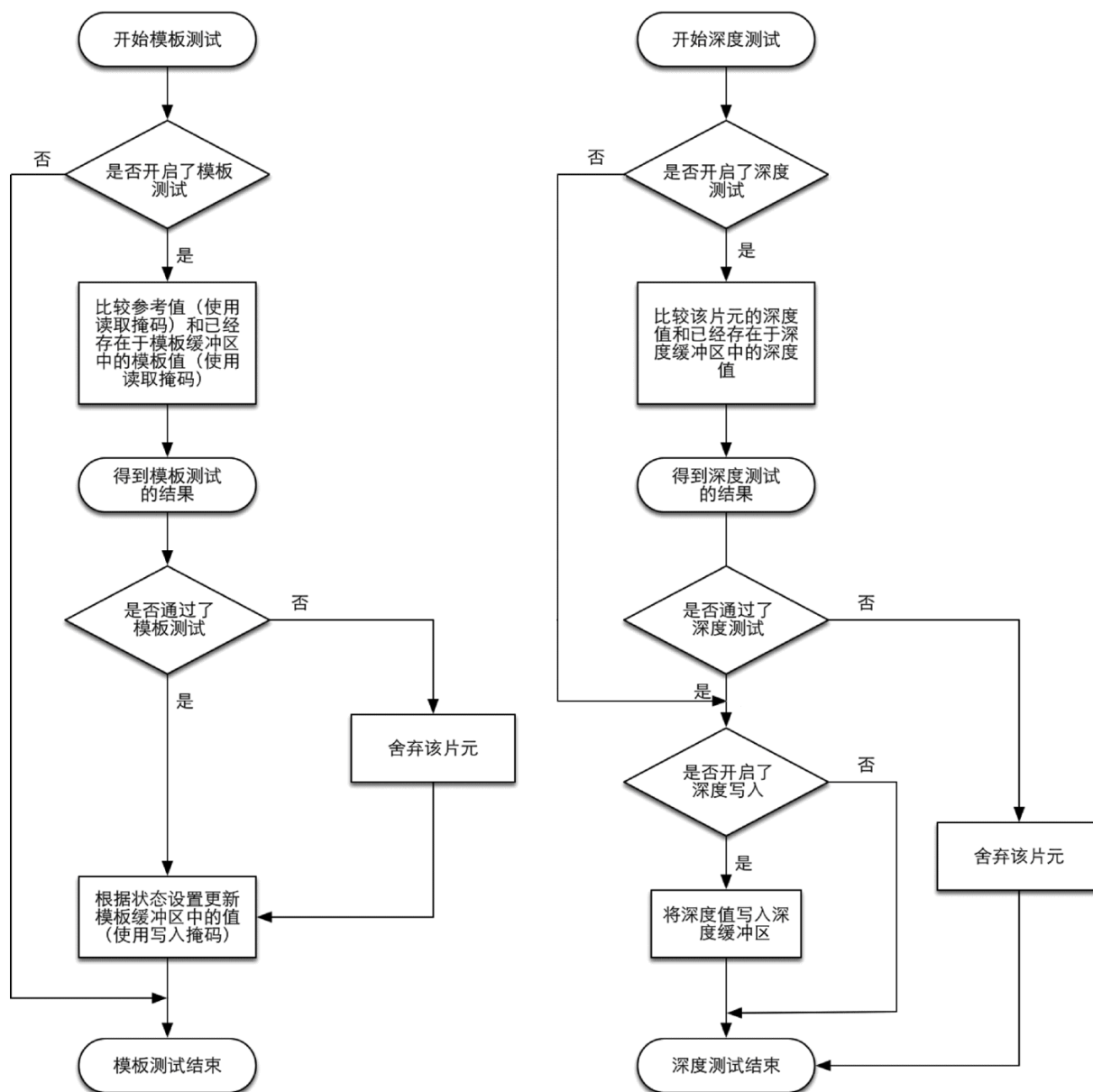
**Poor fragment!** 图2.14给出了简化后的逐片元操作所做的操作。



▲图2.14 逐片元操作阶段所做的操作。只有通过了所有的测试后，新生成的片元才能和颜色缓冲区中已经存在的像素颜色进行混合，最后再写入颜色缓冲区中

测试的过程实际上是个比较复杂的过程，而且不同的图形接口（例如OpenGL和DirectX）的实现细节也不尽相同。这里给出两个最基本的测试——深度测试和模板测试的实现过程。能否理解这些测试过程将关乎读者是否可以理解本书后面章节中提到的渲染队列，尤其是处理透明效果时出现的问题。图2.15给出了深度测试和模板测试的简化流程图。





▲ 图2.15 模板测试和深度测试的简化流程图

我们先来看**模板测试 (Stencil Test)**。与之相关的是模板缓冲 (Stencil Buffer)。实际上，模板缓冲和我们经常听到的颜色缓冲、深度缓冲几乎是一类东西。如果开启了模板测试，GPU会首先读取（使用读取掩码）模板缓冲区中该片元位置的模板值，然后将该值和读取

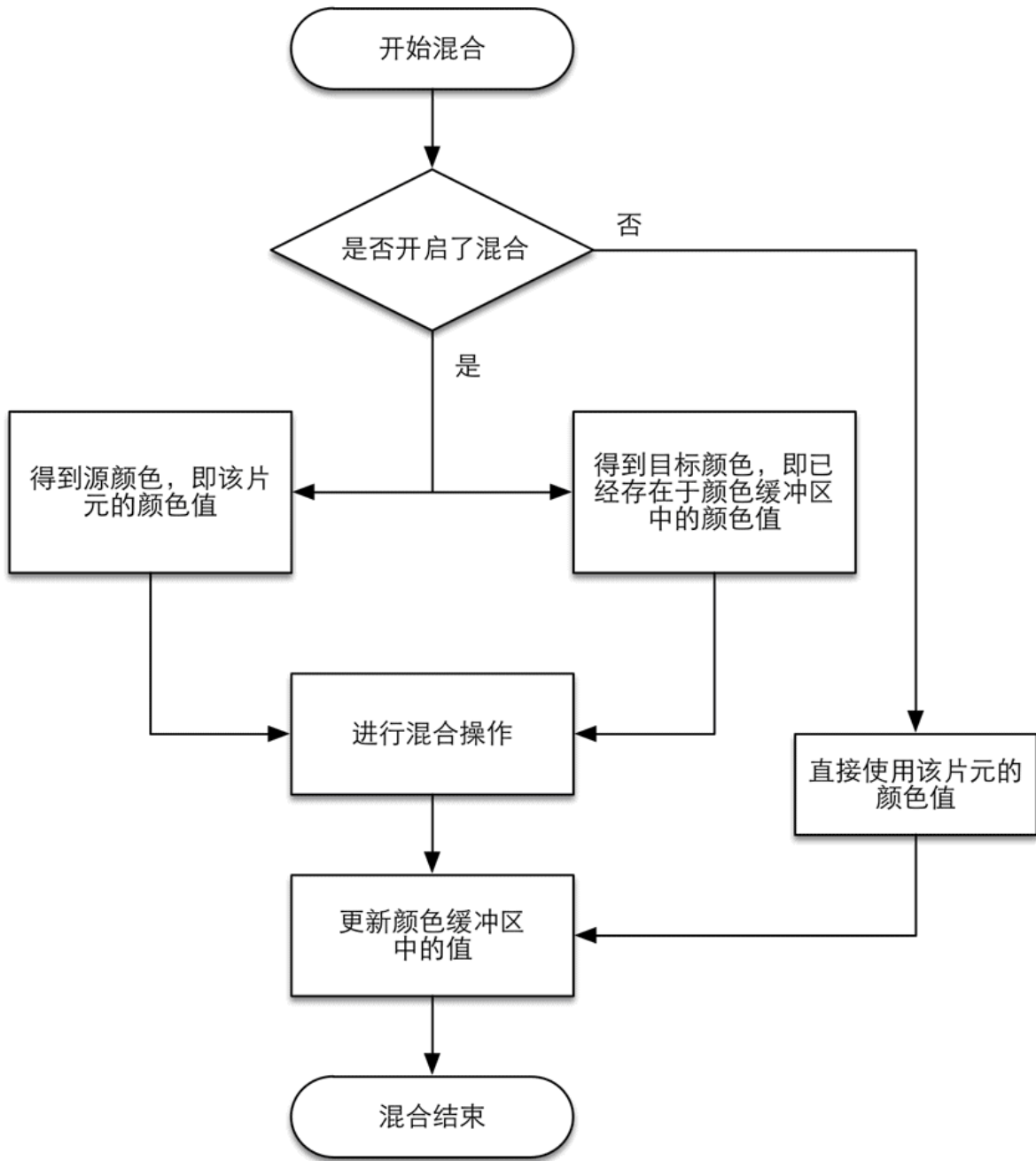
（使用读取掩码）到的参考值（**reference value**）进行比较，这个比较函数可以是由开发者指定的，例如小于时舍弃该片元，或者大于等于时舍弃该片元。如果这个片元没有通过这个测试，该片元就会被舍弃。不管一个片元有没有通过模板测试，我们都可以根据模板测试和下面的深度测试结果来修改模板缓冲区，这个修改操作也是由开发者指定的。开发者可以设置不同结果下的修改操作，例如，在失败时模板缓冲区保持不变，通过时将模板缓冲区中对应位置的值加1等。模板测试通常用于限制渲染的区域。另外，模板测试还有一些更高级的用法，如渲染阴影、轮廓渲染等。

如果一个片元幸运地通过了模板测试，那么它会进行下一个测试——**深度测试（Depth Test）**。相信很多读者都听到过这个测试。这个测试同样是可以高度配置的。如果开启了深度测试，**GPU**会把该片元的深度值和已经存在于深度缓冲区中的深度值进行比较。这个比较函数也是可由开发者设置的，例如小于时舍弃该片元，或者大于等于时舍弃该片元。通常这个比较函数是小于等于的关系，即如果这个片元的深度值大于等于当前深度缓冲区中的值，那么就会舍弃它。这是因为，我们总想只显示出离摄像机最近的物体，而那些被其他物体遮挡的就不需要出现在屏幕上。如果这个片元没有通过这个测试，该片元就会被舍弃。和模板测试有些不同的是，如果一个片元没有通过深度测试，它就没有权利更改深度缓冲区中的值。而如果它通过了测试，开发者还可以指定是否要用这个片元的深度值覆盖掉原有的深度值，这是通过开启/关闭深度写入来做到的。我们在后面的学习中会发现，透明效果和深度测试以及深度写入的关系非常密切。

如果一个幸运的片元通过了上面的所有测试，它就可以自豪地来到**合并**功能的面前。

为什么需要合并？我们要知道，这里所讨论的渲染过程是一个物体接着一个物体画到屏幕上的。而每个像素的颜色信息被存储在一个名为颜色缓冲的地方。因此，当我们执行这次渲染时，颜色缓冲中往往已经有了上次渲染之后的颜色结果，那么，我们是使用这次渲染得到的颜色完全覆盖掉之前的结果，还是进行其他处理？这就是合并需要解决的问题。

对于不透明物体，开发者可以关闭**混合（Blend）**操作。这样片元着色器计算得到的颜色值就会直接覆盖掉颜色缓冲区中的像素值。但对于半透明物体，我们就需要使用混合操作来让这个物体看起来是透明的。图2.16展示了一个简化版的混合操作的流程图。

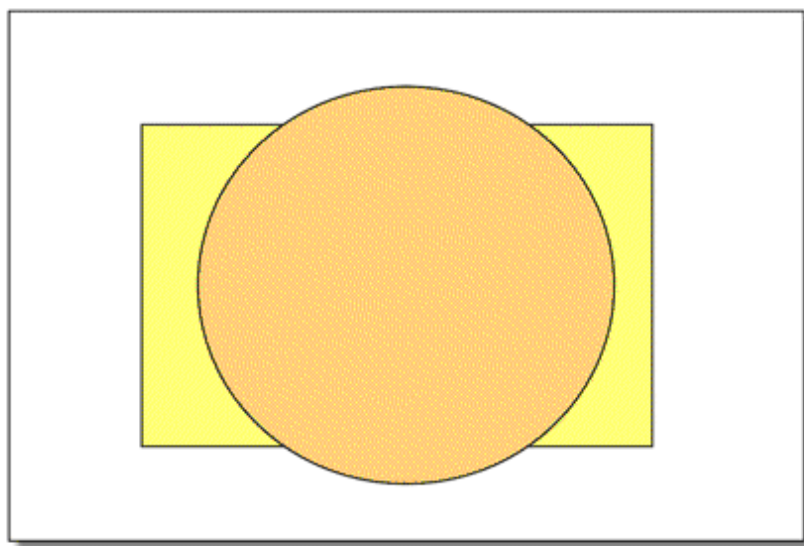


▲图2.16 混合操作的简化流程图

从流程图中我们可以发现，混合操作也是可以高度配置的：开发者可以选择开启/关闭混合功能。如果没有开启混合功能，就会直接使用片元的颜色覆盖掉颜色缓冲区中的颜色，而这也是很多初学者发现

无法得到透明效果的原因（没有开启混合功能）。如果开启了混合，GPU会取出源颜色和目标颜色，将两种颜色进行混合。源颜色指的是片元着色器得到的颜色值，而目标颜色则是已经存在于颜色缓冲区中的颜色值。之后，就会使用一个混合函数来进行混合操作。这个混合函数通常和透明通道息息相关，例如根据透明通道的值进行相加、相减、相乘等。混合很像Photoshop中对图层的操作：每一层图层可以选择混合模式，混合模式决定了该图层和下层图层的混合结果，而我们看到的图片就是混合后的图片。

上面给出的测试顺序并不是唯一的，而且虽然从逻辑上来说这些测试是在片元着色器之后进行的，但对于大多数GPU来说，它们会尽可能在执行片元着色器之前就进行这些测试。这是可以理解的，想象一下，当GPU在片元着色器阶段花了很大力气终于计算出片元的颜色后，却发现这个片元根本没有通过这些检验，也就是说这个片元还是被舍弃了，那之前花费的计算成本全都浪费了！图2.17给出了这样一个场景。



▲图2.17 图示场景中包含了两个对象：球和长方体，绘制顺序是先绘制球（在屏幕上显示为圆），再绘制长方体（在屏幕上显示为长方形）。如果深度测试在片元着色器之后执行，那么在渲染长方体时，虽然它的大部分区域都被遮挡在球的后面，即它所覆盖的绝大部分片元根本无法通过深度测试，但是我们仍然需要对这些片元执行片元着色器，造成了很大的性能浪费

作为一个想充分提高性能的GPU，它会希望尽可能早地知道哪些片元是会被舍弃的，对于这些片元就不需要再使用片元着色器来计算它们的颜色。在Unity给出的渲染流水线中，我们也可以发现它给出的深度测试是在片元着色器之前。这种将深度测试提前执行的技术通常也被称为**Early-Z**技术。希望读者看到这里时不会因此感到困惑。在本书后面的章节中，我们还会继续讨论这个问题。

但是，如果将这些测试提前的话，其检验结果可能会与片元着色器中的一些操作冲突。例如，如果我们在片元着色器进行了透明度测试（我们将在8.3节中具体讲到），而这个片元没有通过透明度测试，我们会在着色器中调用API（例如clip函数）来手动将其舍弃掉。这就导致GPU无法提前执行各种测试。因此，现代的GPU会判断片元着色器中的操作是否和提前测试发生冲突，如果有冲突，就会禁用提前测试。但是，这样也会造成性能上的下降，因为有更多片元需要被处理了。这也是透明度测试会导致性能下降的原因。

当模型的图元经过了上面层层计算和测试后，就会显示到我们的屏幕上。我们的屏幕显示的就是颜色缓冲区中的颜色值。但是，为了避免我们看到那些正在进行光栅化的图元，GPU会使用**双重缓冲**

（**Double Buffering**）的策略。这意味着，对场景的渲染是在幕后发生的，即在**后置缓冲（Back Buffer）**中。一旦场景已经被渲染到了后置缓冲中，GPU就会交换后置缓冲区和**前置缓冲（Front Buffer）**中的内



容，而前置缓冲区是之前显示在屏幕上的图像。由此，保证了我们看到的图像总是连续的。

### 2.3.9 总结

虽然我们上面讲了很多，但其真正的实现过程远比上面讲到的要复杂。需要注意的是，读者可能会发现这里给出的流水线名称、顺序可能和在一些资料上看到的不同。一个原因是由于图像编程接口（如OpenGL和DirectX）的实现不尽相同，另一个原因是GPU在底层可能做了很多优化，例如上面提到的会在片元着色器之前就进行深度测试，以避免无谓的计算。

虽然渲染流水线比较复杂，但Unity作为一个非常出色的平台为我们封装了很多功能。更多时候，我们只需要在一个Unity Shader设置一些输入、编写顶点着色器和片元着色器、设置一些状态就可以达到大部分常见的屏幕效果。这是Unity吸引人的魅力之处，但这样的缺点在于，封装性会导致编程自由度下降，使很多初学者迷失方向，无法掌握其背后的原理，并在出现问题时，往往无法找到错误原因，这是在学习Unity Shader时普遍的遭遇。

渲染流水线几乎和本书所有章节都息息相关，如果读者此时仍然无法完全理解渲染流水线，仍可以继续学习下去。但如果读者在学习过程中发现有些设置或代码无法理解，可以不断查阅本章内容，相信会有更深的理解。

## 2.4 一些容易困惑的地方

在读者学习**Shader**的过程中，会看到一些所谓的专业术语，这些术语的出现频率很高，以至于如果没有对其有基本的认识，会使得初学者总是感到非常困惑。本章的最后将阐述其中的一些术语。

### 2.4.1 什么是**OpenGL/DirectX**

只要读者接触过图像编程，就一定听说过**OpenGL**和**DirectX**，也一定知道这两者之间的竞争关系。**OpenGL**与**DirectX**之间的竞争以及它们与各个硬件生产商之间的纠葛历史很有趣，但很可惜这不在本书的讨论范围。本节的目的在于向读者尽可能通俗地解释，它们到底是什么，又和之前讲到的渲染管线、**GPU**有什么关系。

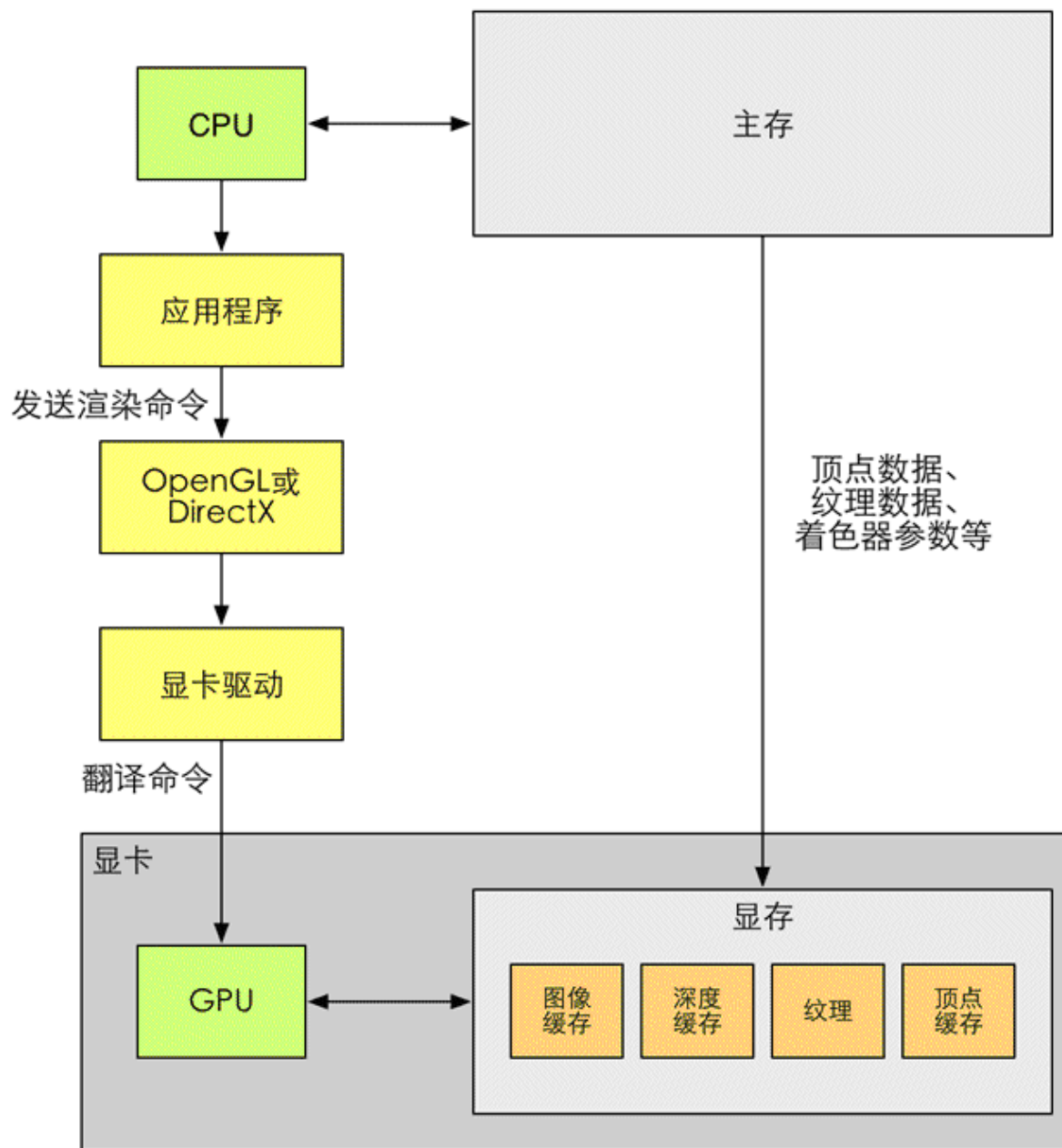
我们花了一整个章节的篇幅来讲述渲染的概念流水线以及**GPU**是如何实现这些流水线的，但如果要开发者直接访问**GPU**是一件非常麻烦的事情，我们可能需要和各种寄存器、显存打交道。而图像编程接口在这些硬件的基础上实现了一层抽象。

**OpenGL**和**DirectX**就是这些图像应用编程接口，这些接口用于渲染二维或三维图形。可以说，这些接口架起了上层应用程序和底层**GPU**的沟通桥梁。一个应用程序向这些接口发送渲染命令，而这些接口会依次向显卡驱动（**Graphics Driver**）发送渲染命令，这些显卡驱动是真正知道如何和**GPU**通信的角色，正是它们把**OpenGL**或者**DirectX**的函数调用翻译成了**GPU**能够听懂的语言，同时它们也负责把纹理等数据转换成**GPU**所支持的格式。一个比喻是，显卡驱动就是显卡的操作系统。图2.18显示了这样的关系。

概括来说，我们的应用程序运行在CPU上。应用程序可以通过调用OpenGL或DirectX的图形接口将渲染所需的数据，如顶点数据、纹理数据、材质参数等数据存储在显存中的特定区域。随后，开发者可以通过图像编程接口发出渲染命令，这些渲染命令也被称为Draw Call，它们将会被显卡驱动翻译成GPU能够理解的代码，进行真正的绘制。

由图2.18可以看出，一个显卡除了有图像处理单元GPU外，还拥有自己的内存，这个内存通常被称为**显存（Video Random Access Memory, VRAM）**。GPU可以在显存中存储任何数据，但对于渲染来说一些数据类型是必需的，例如用于屏幕显示的图像缓冲、深度缓冲等。

因为显卡驱动的存在，几乎所有的GPU都既可以和OpenGL合作，也可以和DirectX一起工作。从显卡的角度出发，实际上它只需要和显卡驱动打交道就可以了。而显卡驱动就好像一个中介者，负责和两方（图像编程接口和GPU）打交道。因此，一个显卡制作商为了让他们的显卡可以同时和OpenGL、DirectX合作，就必须提供支持OpenGL和DirectX接口的显卡驱动。



▲图2.18 CPU、OpenGL/DirectX、显卡驱动和GPU之间的关系

## 2.4.2 什么是HLSL、GLSL、Cg

我们上面讲到了很多可编程的着色器阶段，如顶点着色器、片元着色器等。这些着色器的可编程性在于，我们可以使用一种特定的语

言来编写程序，就好比我们可以用C#来写游戏逻辑一样。

在可编程管线出现之前，为了编写着色器代码，开发者们学习汇编语言。为了给开发者们打开更方便的大门，就出现了更高级的着色语言（**Shading Language**）。着色语言是专门用于编写着色器的，常见的着色语言有DirectX的HLSL（**High Level Shading Language**）、OpenGL的GLSL（**OpenGL Shading Language**）以及NVIDIA的Cg（**C for Graphic**）。HLSL、GLSL、Cg都是“高级（**High-Level**）”语言，但这种高级是相对于汇编语言来说的，而不是像C#相对于C的高级那样。这些语言会被编译成与机器无关的汇编语言，也被称为中间语言（**Intermediate Language, IL**）。这些中间语言再交给显卡驱动来翻译成真正的机器语言，即GPU可以理解的语言。

对于一个初学者来说，一个最常见的问题就是，他应该选择哪种语言？

GLSL的优点在于它的跨平台性，它可以在Windows、Linux、Mac甚至移动平台等多种平台上工作，但这种跨平台性是由于OpenGL没有提供着色器编译器，而是由显卡驱动来完成着色器的编译工作。也就是说，只要显卡驱动支持对GLSL的编译它就可以运行。这种做法的好处在于，由于供应商完全了解自己的硬件构造，他们知道怎样做可以发挥出最大的作用。换句话说，GLSL是依赖硬件，而非操作系统层级的。但这也意味着GLSL的编译结果将取决于硬件供应商。要知道，世界上有很多硬件供应商——NVIDIA、ATI等，他们对GLSL的编译实现不尽相同，这可能会造成编译结果不一致的情况，因为这完全取决于供应商的做法。

而对于HLSL，是由微软控制着色器的编译，就算使用了不同的硬件，同一个着色器的编译结果也是一样的（前提是版本相同）。但也因此支持HLSL的平台相对比较有限，几乎完全是微软自己的产品，如Windows、Xbox 360等。这是因为在其他平台上没有可以编译HLSL的编译器。

Cg则是真正意义上的跨平台。它会根据平台的不同，编译成相应的中间语言。Cg语言的跨平台性很大原因取决于与微软的合作，这也导致Cg语言的语法和HLSL非常相像，Cg语言可以无缝移植成HLSL代码。但缺点是可能无法完全发挥出OpenGL的最新特性。

对于Unity平台，我们同样可以选择使用哪种语言。在Unity Shader中，我们可以选择使用“Cg/HLSL”或者“GLSL”。带引号是因为Unity里的这些着色语言并不是真正意义上的对应的着色语言，尽管它们的语法几乎一样。以Unity Cg为例，你有时会发现有些Cg语法在Unity Shader中是不支持的。关于Unity Shader和真正的Cg/HLSL、GLSL之间的关系我们会在3.6节中讲到。

### 2.4.3 什么是Draw Call

在前面的章节中，我们已经了解了Draw Call的含义。Draw Call本身的含义很简单，就是CPU调用图像编程接口，如OpenGL中的glDrawElements命令或者DirectX中的DrawIndexedPrimitive命令，以命令GPU进行渲染的操作。

一个常见的误区是，Draw Call中造成性能问题的元凶是GPU，认为GPU上的状态切换是耗时的，其实不是的，真正“拖后腿”其实的是



CPU。

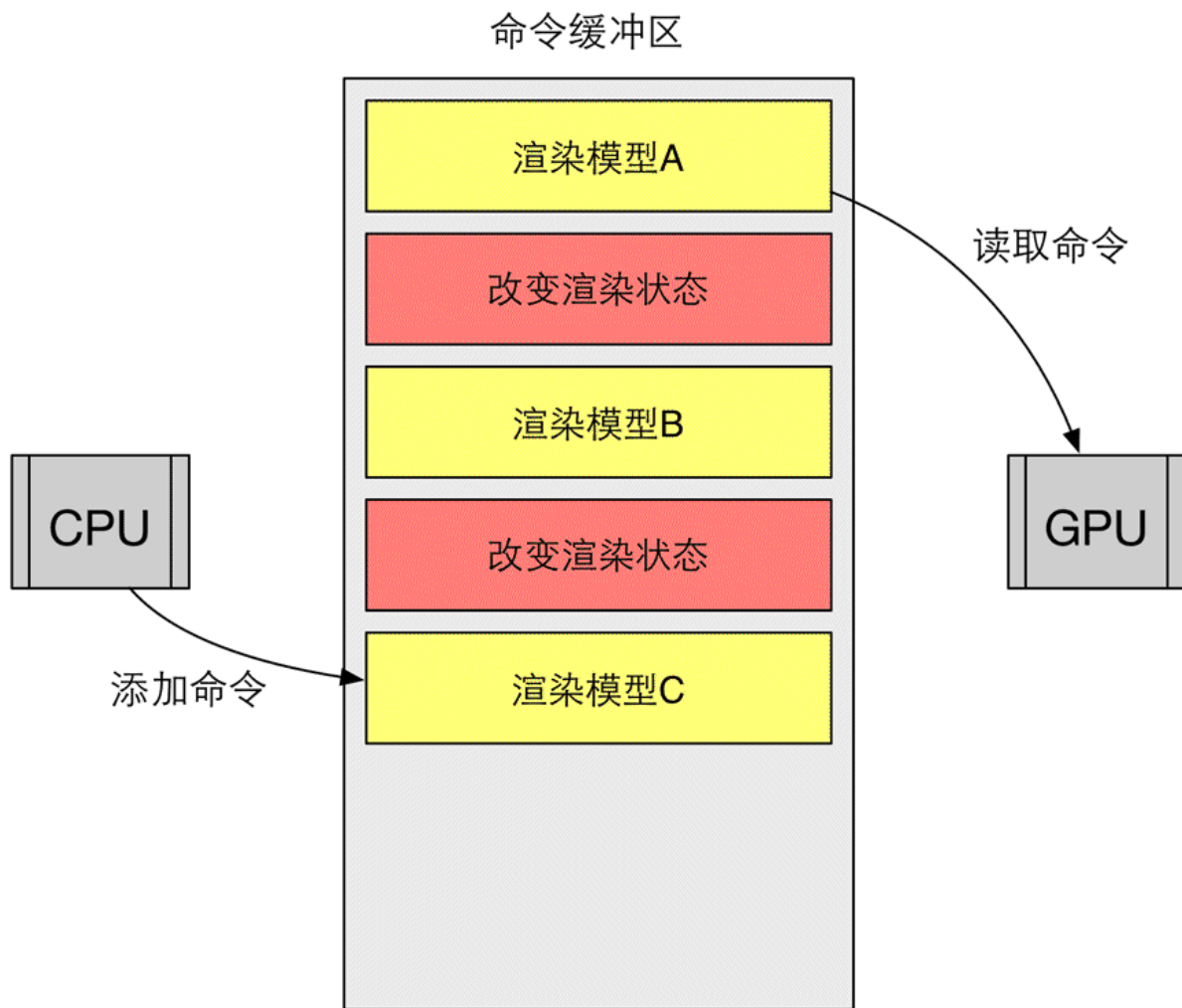
在深入理解Draw Call之前，我们先来看一下CPU和GPU之间的流水线化是怎么实现的，即它们是如何相互独立一起工作的。

### 问题一：CPU和GPU是如何实现并行工作的？

如果没有流水线化，那么CPU需要等到GPU完成上一个渲染任务才能再次发送渲染命令。但这种方法显然会造成效率低下。因此，就像在本章一开头讲到的老王洋娃娃工厂一样，我们需要让CPU和GPU可以并行工作。而解决方法就是使用一个**命令缓冲区（Command Buffer）**。

命令缓冲区包含了一个命令队列，由CPU向其中添加命令，而由GPU从中读取命令，添加和读取的过程是互相独立的。命令缓冲区使得CPU和GPU可以相互独立工作。当CPU需要渲染一些对象时，它可以向命令缓冲区中添加命令，而当GPU完成了上一次的渲染任务后，它就可以从命令队列中再取出一个命令并执行它。

命令缓冲区中的命令有很多种类，而Draw Call是其中一种，其他命令还有改变渲染状态等（例如改变使用的着色器，使用不同的纹理等）。图2.19显示了这样一个例子。



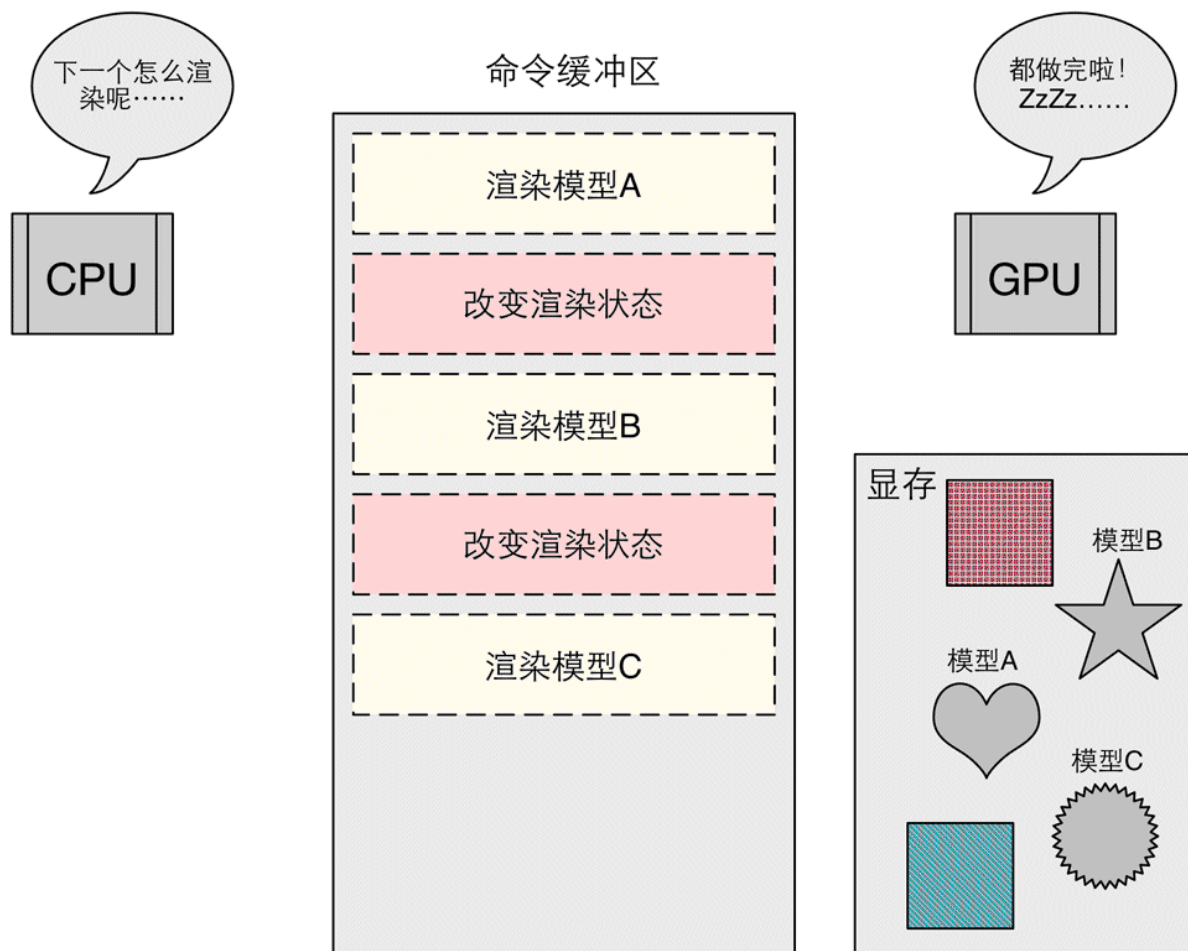
▲图2.19 命令缓冲区。CPU通过图像编程接口向命令缓冲区中添加命令，而GPU从中读取命令并执行。黄色方框内的命令就是Draw Call，而红色方框内的命令用于改变渲染状态。我们使用红色方框来表示改变渲染状态的命令，是因为这些命令往往更加耗时

## 问题二：为什么Draw Call多了会影响帧率？

我们先来做一个实验：请创建10 000个小文件，每个文件的大小为1KB，然后把它们从一个文件夹复制到另一个文件夹。你会发现，尽管这些文件的空间总和不超过10MB，但要花费很长时间。现在，我们再来创建一个单独的文件，它的大小是10MB，然后也把它从一个文件

夹复制到另一个文件夹。而这次复制的时间却少很多！这是为什么呢？明明它们所包含的内容大小是一样的。原因在于，每一个复制动作需要很多额外的操作，例如分配内存、创建各种元数据等。如你所见，这些操作将造成很多额外的性能开销，如果我们复制了很多小文件，那么这个开销将会很大。

渲染的过程虽然和上面的实验有很大不同，但从感性角度上是很类似的。在每次调用**Draw Call**之前，CPU需要向GPU发送很多内容，包括数据、状态和命令等。在这一阶段，CPU需要完成很多工作，例如检查渲染状态等。而一旦CPU完成了这些准备工作，GPU就可以开始本次的渲染。GPU的渲染能力是很强的，渲染200个还是2 000个三角网格通常没有什么区别，因此渲染速度往往快于CPU提交命令的速度。如果**Draw Call**的数量太多，CPU就会把大量时间花费在提交**Draw Call**上，造成CPU的过载。图2.20显示了这样一个例子。



▲图2.20 命令缓冲区中的虚线方框表示GPU已经完成的命令。此时，命令缓冲区中没有可以执行的命令了，GPU处于空闲状态，而CPU还没有准备好下一个渲染命令

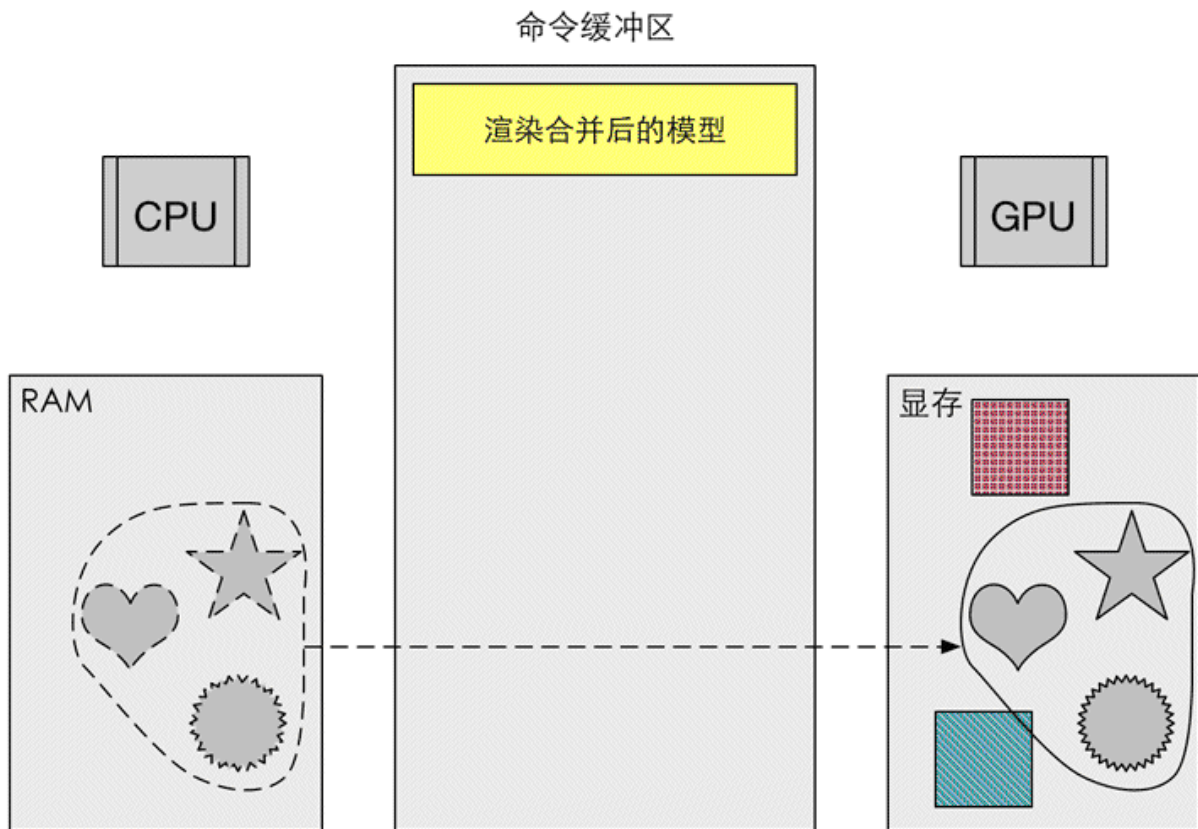
### 问题三：如何减少Draw Call？

尽管减少Draw Call的方法有很多，但我们这里仅讨论使用**批处理（Batching）**的方法。

我们讲过，提交大量很小的Draw Call会造成CPU的性能瓶颈，即CPU把时间都花费在准备Draw Call的工作上了。那么，一个很显然的

优化想法就是把很多小的DrawCall合并成一个大的Draw Call，这就是批处理的思想。图2.21显示了批处理所做的工作。

需要注意的是，由于我们需要在CPU的内存中合并网格，而合并的过程是需要消耗时间的。因此，批处理技术更加适合于那些静态的物体，例如不会移动的大地、石头等，对于这些静态物体我们只需要合并一次即可。当然，我们也可以对动态物体进行批处理。但是，由于这些物体是不断运动的，因此每一帧都需要重新进行合并然后再发送给GPU，这对空间和时间都会造成一定的影响。



▲图2.21 利用批处理，CPU在RAM把多个网格合并成一个更大的网格，再发送给GPU，然后在一个Draw Call中渲染它们。但要注意的是，使用批处理合并的网格将会使用同一种渲染状态。也就是说，如果网格之间需要使用不同的渲染状态，那么就无法使用批处理技术

在游戏开发过程中，为了减少Draw Call的开销，有两点需要注意。

(1) 避免使用大量很小的网格。当不可避免地需要使用很小的网格结构时，考虑是否可以合并它们。

(2) 避免使用过多的材质。尽量在不同的网格之间共用同一个材质。

在本书的16.4节，我们会继续阐述如何在Unity中利用批处理技术来进行优化。

#### 2.4.4 什么是固定管线渲染

**固定函数的流水线（Fixed-Function Pipeline）**，也简称为固定管线，通常是指在较旧的GPU上实现的渲染流水线。这种流水线只给开发者提供一些配置操作，但开发者没有对流水线阶段的完全控制权。

固定管线通常提供了一系列接口，这些接口包含了一个函数入口点（Function Entry Points）集合，这些函数入口点会匹配GPU上的一个特定的逻辑功能。开发者们通过这些接口来控制渲染流水线。换句话说，固定渲染管线是只可配置的管线。一个形象的比喻是，我们在使用固定管线进行渲染时，就好像在控制电路上的多个开关，我们可以选择打开或者关闭一个开关，但永远无法控制整个电路的排布。

随着时代的发展，GPU流水线越来越朝着更高的灵活性和可控性方向发展，可编程渲染管线应运而生。我们在上面看到了许多可编程的流水线阶段，如顶点着色器、片元着色器，这些可编程的着色器阶



段可以说是GPU进化最重要的贡献。表2.1给出了3种最常见的图像接口从固定管线向可编程管线进化的版本。

表2.1     3种图像接口从固定管线向可编程管线进化的版本

3D API	最后支持固定管线的版本	第一个支持可编程管线的版本
OpenGL	1.5	2.0
OpenGL ES	1.1	2.0
DirectX	7.0	8.0

在GPU发展的过程中，为了继续提供固定管线的接口抽象，一些显卡驱动的开发者们使用了更加通用的着色架构，即使用可编程的管线来模拟固定管线。这是为了在提供可编程渲染管线的同时，可以让那些已经熟悉了固定管线的开发者们继续使用固定管线进行渲染。例如，OpenGL 2.0在没有真正的固定管线的硬件支持下，依靠系统的可编程管线功能来模仿固定管线的处理过程。但随着GPU的发展，固定管线已经逐渐退出历史舞台。例如，OpenGL 3.0是最后既支持可编程管线又完全支持固定管线编程接口的版本，在OpenGL 3.2中，Core Profile就完全移除了固定管线的概念。

因此，如果读者不是为了对较旧的设备进行兼容，不建议继续使用固定管线的渲染方式。

## 2.5 那么，你明白什么是Shader了吗

我们之所以要花很大篇幅来讲述GPU的渲染流水线，是因为Shader所在的阶段就是渲染流水线的一部分，更具体来说，Shader就是：

- GPU流水线上一些可高度编程的阶段，而由着色器编译出来的最终代码是会在GPU上运行的（对于固定管线的渲染来说，着色器有时等同于一些特定的渲染设置）；
- 有一些特定类型的着色器，如顶点着色器、片元着色器等；
- 依靠着色器我们可以控制流水线中的渲染细节，例如用顶点着色器来进行顶点变换以及传递数据，用片元着色器来进行逐像素的渲染。

但同时，我们也要明白，要得到出色的游戏画面是需要包括Shader在内的所有渲染流水线阶段的共同参与才可完成：设置适当的渲染状态，使用合适的混合函数，开启还是关闭深度测试/深度写入等。

Unity作为一个出色的编辑工具，为我们提供了一个既可以方便地编写着色器，同时又可设置渲染状态的地方：Unity Shader。在下一章中，我们将真正走进Unity Shader的世界。

## 2.6 扩展阅读

如果读者对渲染流水线的细节感兴趣，可以阅读更多的资料。托马斯在他们的著作<sup>[1]</sup>中给出了很多有关实时渲染的内容，这本书被誉为图形学中的圣经。如果你仍然觉得本书讲解的Draw Call不够形象生

动，西蒙在他的文章中给出了很多动态的演示效果，而且值得注意的是，西蒙本人是一位美术工作者。为什么需要批处理，什么时候需要批处理等更多关于批处理的内容，可以在NVIDIA所做的一次报告<sup>[2]</sup>中找到更多的答案。如果读者对OpenGL和DirectX的渲染流水线的实现细节感兴趣，那么阅读它们的文档（[https://www.opengl.org/wiki/Rendering\\_Pipeline\\_Overview](https://www.opengl.org/wiki/Rendering_Pipeline_Overview)，[https://msdn.microsoft.com/en-us/library/windows/desktop/ff476882\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ff476882(v=vs.85).aspx)）是一个非常好的途径。

[1] Akenine-Möller T, Haines E, Hoffman N. Real-time rendering[M]. CRC Press, 2008.

[2] Wloka M. Batch, Batch, Batch: What does it really mean? [C]//Presentation at game developers conference. 2003.

## 第3章 Unity Shader基础

通过前面的学习内容我们已经知道，Shader并不是什么神秘的东西，它们其实就是渲染流水线中的某些特定阶段，如顶点着色器阶段、片元着色器阶段等。

在没有Unity这类编辑器的情况下，如果我们想要对某个模型设置渲染状态，可能需要类似下面的代码：

```
// 初始化渲染设置
void Initialization() {
    // 从硬盘上加载顶点着色器的代码
    string vertexShaderCode =
LoadShaderFromFile(VertexShader.shader);
    // 从硬盘上加载片元着色器的代码
    string fragmentShaderCode =
LoadShaderFromFile(FragmentShader.shader);
    // 把顶点着色器加载到GPU中
    LoadVertexShaderFromString(vertexShaderCode);
    // 把片元着色器加载到GPU中
    LoadFragmentShaderFromString(fragmentShaderCode);

    // 设置名为"vertexPosition"的属性的输入，即模型顶点坐标
    SetVertexShaderProperty("vertexPosition", vertices);
    // 设置名为"MainTex"的属性的输入，someTexture是某张已加载的纹理
    SetVertexShaderProperty("MainTex", someTexture);
    // 设置名为"MVP"的属性的输入，MVP是之前由开发者计算好的变换矩阵
    SetVertexShaderProperty("MVP", MVP);

    // 关闭混合
    Disable(Blend);
    // 设置深度测试
    Enable(ZTest);
    SetZTestFunction(LessOrEqual);

    // 其他设置
    ...
}
```

```
// 每一帧进行渲染
void OnRendering() {
    // 调用渲染命令
    DrawCall();
    // 当涉及多种渲染设置时，我们可能还需要在这里改变各种渲染设置
    ...
}
```

### VertexShader.shader:

```
// 输入：顶点位置、纹理、MVP变换矩阵
in float3 vertexPosition;
in sampler2D MainTex;
in Matrix4x4 MVP;

// 输出：顶点经过MVP变换后的位置
out float4 position;

void main() {
    // 使用MVP对模型顶点坐标进行变换
    position = MVP * vertexPosition;
}
```

### FragmentShader.shader:

```
// 输入：VertexShader输出的position、经过光栅化程序插值后的该片元对应的
position
in float4 position;

// 输出：该片元的颜色值
out float4 fragColor;

void main() {
    // 将片元颜色设为白色
    fragColor = float4(1.0, 1.0, 1.0, 1.0);
}
```

上述伪代码仅仅是简化后的版本，当渲染的模型数目、需要调整的着色器属性不断增多时，上述过程将变得更加复杂和冗长。而且，当涉及透明物体等多物体的渲染时，如果没有编辑器的帮助，我们要非常小心如渲染顺序等问题。

Unity的出现改善了上面的状况。它提供了一个地方能够让开发者更加轻松地管理着色器代码以及渲染设置（如开启/关闭混合、深度测试、设置渲染顺序等），而不需要像上面的伪代码一样，管理多个文件和函数等。Unity提供的这个“方便的地方”，就是Unity Shader。

## 3.1 Unity Shader概述

那么如何充分利用Unity Shader来为我们的游戏增光添彩呢？

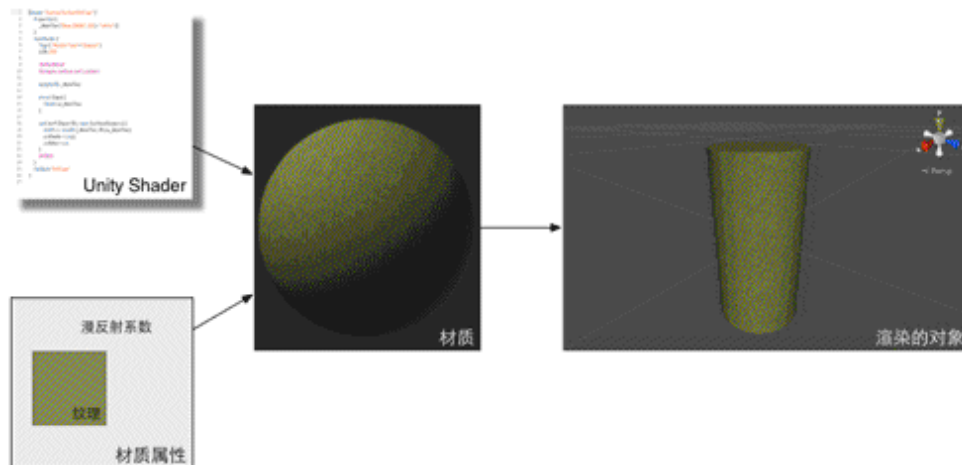
### 3.1.1 一对好兄弟：材质和Unity Shader

总体来说，在Unity中我们需要配合使用材质（**Material**）和Unity Shader才能达到需要的效果。一个最常见的流程是：

- （1）创建一个材质；
- （2）创建一个Unity Shader，并把它赋给上一步中创建的材质；
- （3）把材质赋给要渲染的对象；
- （4）在材质面板中调整Unity Shader的属性，以得到满意的效果。

图3.1显示了Unity Shader和材质是如何一起工作来控制物体的渲染的。





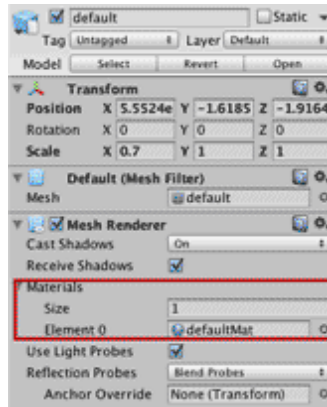
▲图3.1 Unity Shader和材质。首先创建需要的Unity Shader和材质，然后把Unity Shader赋给材质，并在材质面板上调整属性（如使用的纹理、漫反射系数等）。最后，将材质赋给相应的模型来查看最终的渲染效果

可以发现，Unity Shader定义了渲染所需的各种代码（如顶点着色器和片元着色器）、属性（如使用哪些纹理等）和指令（渲染和标签设置等），而材质则允许我们调节这些属性，并将其最终赋给相应的模型。

### 3.1.2 Unity中的材质

Unity中的材质需要结合一个GameObject的Mesh或者Particle Systems组件来工作。它决定了我们的游戏对象看起来是什么样子的（这当然也需要Unity Shader的配合）。

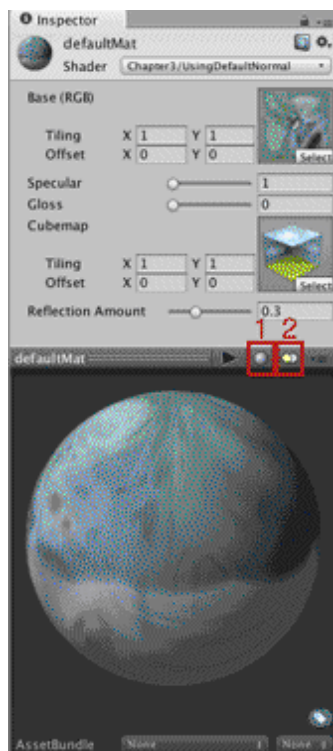
为了创建一个新的材质，我们可以在Unity的菜单栏中选择Assets -> Create -> Material来创建，也可以直接在Project视图中右击 -> Create -> Material来创建。当创建了一个材质后，就可以把它赋给一个对象。这可以通过把材质直接拖曳到Scene视图中的对象上来实现，或者在该对象的Mesh Renderer组件中直接赋值，如图3.2所示。



▲ 图3.2 将材质直接拖曳到模型的Mesh Renderer组件中

在Unity 5.x版本中，默认情况下，一个新建的材质将使用Unity内置的Standard Shader，这是一种基于物理渲染的着色器，我们将在第18章中讲到。

对于美术人员来说，材质是他们十分熟悉的一种事物。Unity的材质和许多建模软件（如Cinema 4D、Maya等）中提供的材质功能类似，它们都提供了一个面板来调整材质的各个参数。这种可视化的方法使得开发者不再需要自行在代码中设置和改变渲染所需的各种参数，如图3.3所示。



▲ 图3.3 材质提供了一种可视化的方式来调整着色器中使用的参数



#### 提示

单击图标“1”可变换面板中使用的基础模型种类，Unity支持球、立方体、圆柱体等多种基础模型；单击图标“2”可变换面板中使用的光照。

### 3.1.3 Unity中的Shader

为了和前面通用的Shader语义进行区分，我们把Unity中的Shader文件统称为**Unity Shader**。这是因为，Unity Shader和我们之前提及的渲染管线的Shader有很大不同，我们会在3.6.2节中进行更加详细的解释。

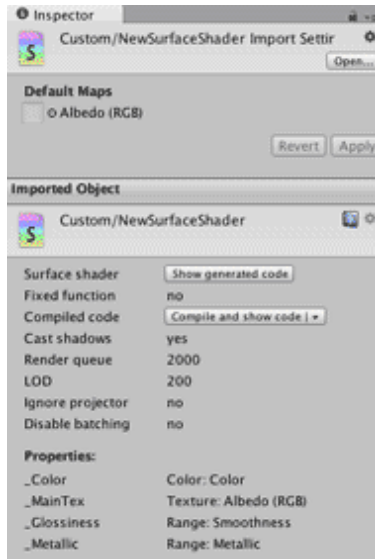
为了创建一个新的Unity Shader，我们可以在Unity的菜单栏中选择 *Assets -> Create -> Shader* 来创建，也可以直接在Project视图中右击 -> *Create -> Shader* 来创建。在Unity 5.2及以上版本中，Unity一共提供了4种Unity Shader模板供我们选择——*Standard Surface Shader*，*Unlit Shader*，*Image Effect Shader*以及*Compute Shader*。其中，*Standard Surface Shader*会产生一个包含了标准光照模型（使用了Unity 5中新添加的基于物理的渲染方法，详见第18章）的表面着色器模板，*Unlit Shader*则会产生一个不包含光照（但包含雾效）的基本的顶点/片元着色器，*Image Effect Shader*则为我们实现各种屏幕后处理效果（详见第12章）提供了一个基本模板。最后，*Compute Shader*会产生一种特殊的Shader文件，这类Shader旨在利用GPU的并行性来进行一些与常规渲染流水线无关的计算，而这不在本书的讨论范围内，读者可以在Unity手册的**Compute Shader**一文（<http://docs.unity3d.com/Manual/ComputeShaders.html>）中找到更多的介绍。总体来说，*Standard Surface Shader*为我们提供了典型的表面着色器的实现方法，但本书的重点在于如何在Unity中编写顶点/片元着色器，因此在后续的学习中，我们通常会使用*Unlit Shader*来生成一个基本的顶点/片元着色器模板。

一个单独的Unity Shader是无法发挥任何作用的，它必须和材质结合起来，才能发生神奇的“化学反应”！为此，我们可以在材质面板最上方的下拉菜单中选择需要使用的Unity Shader。当选择完毕后，材质面板中就会出现该Unity Shader可用的各种属性。这些属性可以是颜色、纹理、浮点数、滑动条（限制了范围的浮点数）、向量等。当我们把材质赋给场景中的一个对象时，就可以看到调整属性所发生的视觉变化。

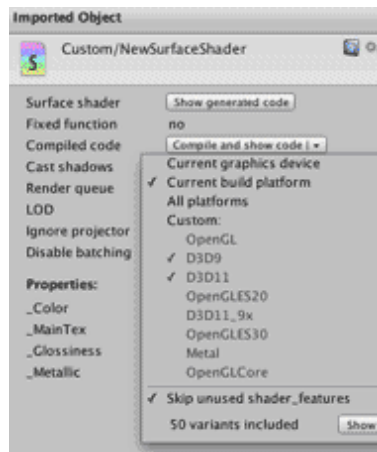
Unity Shader本质上就是一个文本文件。和Unity中的很多外部文件类似，Unity Shader也有导入设置（Import Settings）面板，在**Project**视图中选中某个Unity Shader即可看到。在Unity 5.2版本中，Unity Shader的导入设置面板如图3.4所示。

在该面板上，我们可以在**Default Maps**中指定该Unity Shader使用的默认纹理。当任何材质第一次使用该Unity Shader时，这些纹理就会自动被赋予到相应的属性上。在下方的面板中，Unity会显示出和该Unity Shader相关的信息，例如它是否是一个表面着色器（Surface Shader）、是否是一个固定函数着色器（Fixed Function Shader）等，还有一些信息是和我们Unity Shader中的标签设置（详见3.3.3节）有关，例如是否会投射阴影、使用的渲染队列、LOD值等。

对于表面着色器（详见3.4.1节）来说，我们可以通过单击**Show generated code**按钮来打开一个新的文件，在该文件里将显示Unity在背后为该表面着色器生成的顶点/片元着色器。这可以方便我们对这些生成的代码进行修改（需要复制到一个新的Unity Shader中才可保存）和研究。同样地，如果该Unity Shader是一个固定函数着色器，在**Fixed function**的后面也会出现一个**Show generated code**按钮，来让我们查看该固定函数着色器生成的顶点/片元着色器。**Compile and show code**下拉列表可以让开发者检查该Unity Shader针对不同图像编程接口（例如OpenGL、D3D9、D3D11等）最终编译成的Shader代码，如图3.5所示。直接单击该按钮可以查看生成的底层的汇编指令。我们可以利用这些代码来分析和优化着色器。



▲图3.4 Unity Shader的导入设置面板



▲图3.5 Compile and show code下拉列表

除此之外，Unity Shader的导入面板还可以方便地查看其使用的渲染队列（Render queue）、是否关闭批处理（Disable batching）、属性列表（Properties）等信息。

## 3.2 Unity Shader的基础：ShaderLab



“计算机科学中的任何问题都可以通过增加一层抽象来解决。”—— 大卫·惠勒

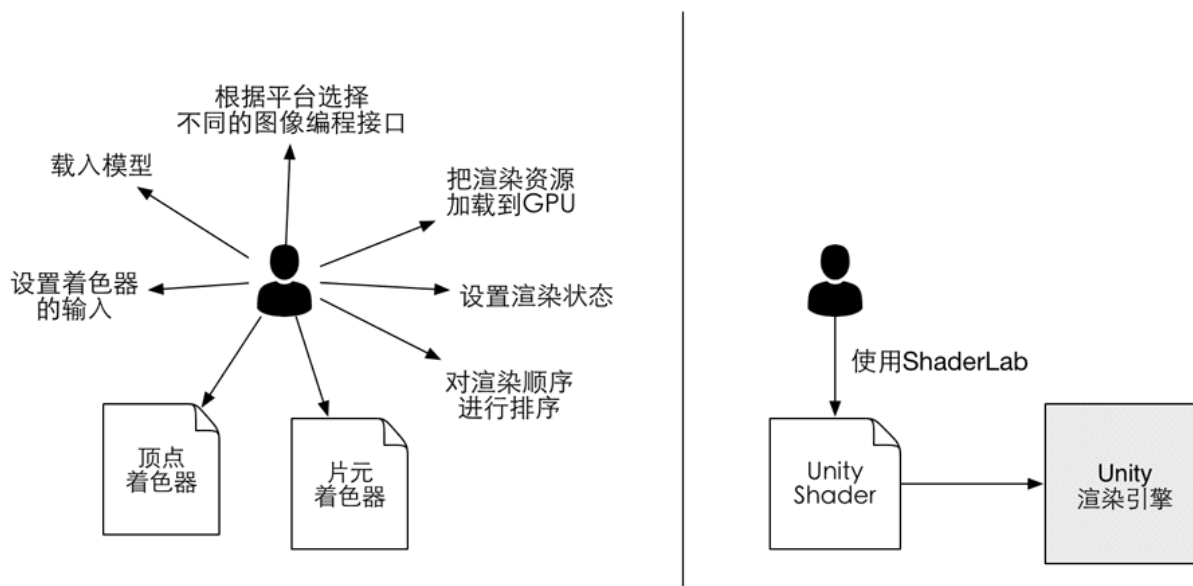
学习和编写着色器的过程一直是一个学习曲线很陡峭的过程。通常情况下，为了自定义渲染效果往往需要和很多文件和设置打交道，这些过程很容易消磨掉初学者的耐心。而且，一些细节问题也往往需要开发者花费较多的时间去解决。

Unity为了解决上述问题，为我们提供了一层抽象——Unity Shader。而我和这层抽象打交道的途径就是使用Unity提供的一种专门为Unity Shader服务的语言——**ShaderLab**。

## 什么是ShaderLab?

"ShaderLab is a friend you can afford."——尼古拉斯·弗朗西斯（Nicholas Francis），Unity前首席运营官（COO）和联合创始人之一。

Unity Shader是Unity为开发者提供的高层级的渲染抽象层。图3.6显示了这样的抽象。Unity希望通过这种方式来让开发者更加轻松地控制渲染。



▲图3.6 Unity Shader为控制渲染过程提供了一层抽象。如果没有使用Unity Shader（左图），开发者需要和很多文件和设置打交道，才能让画面呈现出想要的效果；而在Unity Shader的帮助下（右图），开发者只需要使用ShaderLab来编写Unity Shader文件就可以完成所有的工作

在Unity中，所有的Unity Shader都是使用ShaderLab来编写的。ShaderLab是Unity提供的编写Unity Shader的一种说明性语言。它使用了一些嵌套在花括号内部的语义（**syntax**）来描述一个Unity Shader文件的结构。这些结构包含了许多渲染所需的数据，例如`Properties`语句块中定义了着色器所需的各种属性，这些属性将会出现在材质面板中。从设计上来说，ShaderLab类似于CgFX和Direct3D Effects（.FX）语言，它们都定义了要显示一个材质所需的所有东西，而不仅仅是着色器代码。

一个Unity Shader的基础结构如下所示：

```
Shader "ShaderName" {
    Properties {
        // 属性
    }
}
```

```
SubShader {  
    // 显卡A使用的子着色器  
}  
SubShader {  
    // 显卡B使用的子着色器  
}  
Fallback "VertexLit"  
}
```

Unity在背后会根据使用的平台来把这些结构编译成真正的代码和Shader文件，而开发者只需要和Unity Shader打交道即可。

## 3.3 Unity Shader的结构

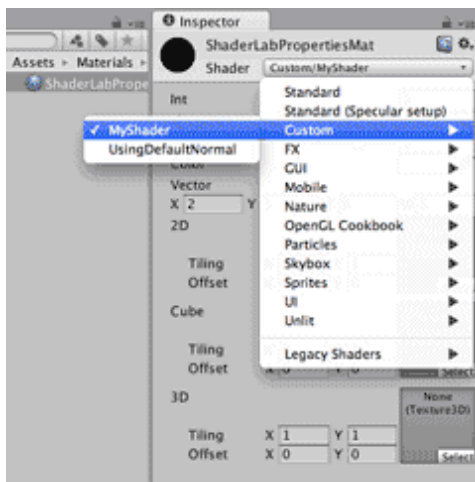
在上一节的伪代码中我们见到了一些ShaderLab的语义，如 *Properties*、*SubShader*、*Fallback*等。这些语义定义了Unity Shader的结构，从而帮助Unity分析该Unity Shader文件，以便进行正确的编译。在下面，我们会解释这些基础的语义含义和用法。

### 3.3.1 给我们的Shader起个名字

每个Unity Shader文件的第一行都需要通过*Shader*语义来指定该Unity Shader的名字。这个名字由一个字符串来定义，例如“*MyShader*”。当为材质选择使用的Unity Shader时，这些名称就会出现在材质面板的下拉列表里。通过在字符串中添加斜杠（“/”），可以控制Unity Shader在材质面板中出现的位置。例如：

```
Shader "Custom/MyShader" {    }
```

那么这个Unity Shader在材质面板中的位置就是：*Shader -> Custom -> MyShader*，如图3.7所示。



▲ 图3.7 在Unity Shader的名称定义中利用斜杠来组织在材质面板中的位置

### 3.3.2 材质和Unity Shader的桥梁: Properties

*Properties*语义块中包含了一系列属性（**property**），这些属性将会出现在材质面板中。

*Properties*语义块的定义通常如下：

```
Properties {
    Name ("display name", PropertyType) = DefaultValue
    Name ("display name", PropertyType) = DefaultValue
    // 更多属性
}
```

开发者们声明这些属性是为了在材质面板中能够方便地调整各种材质属性。如果我们需要在Shader中访问它们，就需要使用每个属性的名字（**Name**）。在Unity中，这些属性的名字通常由一个下划线开始。显示的名称（**display name**）则是出现在材质面板上的名字。我们需要为每个属性指定它的类型（**PropertyType**），常见的属性类型如表3.1所示。除此之外，我们还需要为每个属性指定一个默认值，在我们第

一次把该Unity Shader赋给某个材质时，材质面板上显示的就是这些默认值。

表3.1 Properties语义块支持的属性类型

属性类型	默认值的定义语法	例子
Int	number	<code>_Int ("Int", Int) = 2</code>
Float	number	<code>_Float ("Float", Float) = 1.5</code>
Range(min, max)	number	<code>_Range("Range", Range(0.0, 5.0)) = 3.0</code>
Color	<code>(number,number,number,number)</code>	<code>_Color ("Color", Color) = (1,1,1,1)</code>
Vector	<code>(number,number,number,number)</code>	<code>_Vector ("Vector", Vector) = (2, 3, 6, 1)</code>
2D	<code>"defaulttexture" {}</code>	<code>_2D ("2D", 2D) = "" {}</code>
Cube	<code>"defaulttexture" {}</code>	<code>_Cube ("Cube", Cube) = "white" {}</code>
3D	<code>"defaulttexture" {}</code>	<code>_3D ("3D", 3D) = "black" {}</code>

对于**Int**、**Float**、**Range**这些数字类型的属性，其默认值就是一个单独的数字；对于**Color**和**Vector**这类属性，默认值是用圆括号包围的

一个四维向量；对于**2D**、**Cube**、**3D**这3种纹理类型，默认值的定义稍微复杂，它们的默认值是通过一个字符串后跟一个花括号来指定的，其中，字符串要么是空的，要么是内置的纹理名称，如“white”“black”“gray”或者“bump”。花括号的用处原本是由于指定一些纹理属性的，例如在Unity 5.0以前的版本中，我们可以通过**TexGen CubeReflect**、**TexGen CubeNormal**等选项来控制固定管线的纹理坐标的生成。但在Unity 5.0以后的版本中，这些选项被移除了，如果我们需要类似的功能，就需要自己在顶点着色器中编写计算相应纹理坐标的代码。

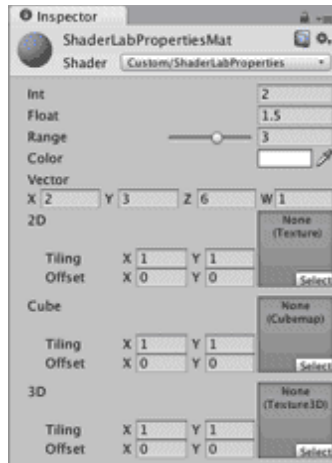
下面的代码给出了一个展示所有属性类型的例子：

```
Shader "Custom/ShaderLabProperties" {
    Properties {
        // Numbers and Sliders
        _Int ("Int", Int) = 2
        _Float ("Float", Float) = 1.5
        _Range("Range", Range(0.0, 5.0)) = 3.0
        // Colors and Vectors
        _Color ("Color", Color) = (1,1,1,1)
        _Vector ("Vector", Vector) = (2, 3, 6, 1)
        // Textures
        _2D ("2D", 2D) = "" {}
        _Cube ("Cube", Cube) = "white" {}
        _3D ("3D", 3D) = "black" {}
    }

    FallBack "Diffuse"
}
```

图3.8给出了上述代码在材质面板中的显示结果。





▲ 图3.8 不同属性类型在材质面板中的显示结果

有时，我们想要在材质面板上显示更多类型的变量，例如使用布尔变量来控制Shader中使用哪种计算。Unity允许我们重载默认的材质编辑面板，以提供更多自定义的数据类型。我们在本书资源的材质 *Assets -> Materials -> Chapter3 -> RedifyMat* 中提供了这样一个简单的例子，这个例子参考了官方手册的**Custom Shader GUI**一文（<http://docs.unity3d.com/Manual/SL-CustomShaderGUI.html>）中的代码。

为了在Shader中可以访问到这些属性，我们需要在Cg代码片中定义和这些属性类型相匹配的变量。需要说明的是，即使我们不在 *Properties* 语义块中声明这些属性，也可以直接在Cg代码片中定义变量。此时，我们可以通过脚本向Shader中传递这些属性。因此，*Properties* 语义块的作用仅仅是为了让这些属性可以出现在材质面板中。

### 3.3.3 重量级成员：SubShader

每一个Unity Shader文件可以包含多个*SubShader*语义块，但最少要有一个。当Unity需要加载这个Unity Shader时，Unity会扫描所有的*SubShader*语义块，然后选择第一个能够在目标平台上运行的*SubShader*。如果都不支持的话，Unity就会使用*Fallback*语义指定的Unity Shader。

Unity提供这种语义的原因在于，不同的显卡具有不同的能力。例如，一些旧的显卡仅能支持一定数目的操作指令，而一些更高级的显卡可以支持更多的指令数，那么我们希望在旧的显卡上使用计算复杂度较低的着色器，而在高级的显卡上使用计算复杂度较高的着色器，以便提供更出色的画面。

*SubShader*语义块中包含的定义通常如下：

```
SubShader {  
    // 可选的  
    [Tags]  
  
    // 可选的  
    [RenderSetup]  
  
    Pass {  
    }  
    // Other Passes  
}
```

*SubShader*中定义了一系列*Pass*以及可选的状态（[RenderSetup]）和标签（[Tags]）设置。每个*Pass*定义了一次完整的渲染流程，但如果*Pass*的数目过多，往往会造成渲染性能的下降。因此，我们应尽量使用最小数目的*Pass*。状态和标签同样可以在*Pass*声明。不同的是，*SubShader*中的一些标签设置是特定的。也就是说，这些标签设置和*Pass*中使用的标签是不一样的。而对于状态设置来说，其使用的语法

是相同的。但是，如果我们在*SubShader*进行了这些设置，那么将会用于所有的*Pass*。

- 状态设置

ShaderLab提供了一系列渲染状态的设置指令，这些指令可以设置显卡的各种状态，例如是否开启混合/深度测试等。表3.2给出了ShaderLab中常见的渲染状态设置选项。

表3.2 常见的渲染状态设置选项

状 态 名 称	设 置 指 令	解 释
Cull	Cull Back   Front   Off	设置剔除模式：剔除背面/正面/关闭剔除
ZTest	ZTest Less Greater   LEqual   GEqual   Equal   NotEqual   Always	设置深度测试时使用的函数
ZWrite	ZWrite On   Off	开启/关闭深度写入
Blend	Blend SrcFactor DstFactor	开启并设置混合模式

当在*SubShader*块中设置了上述渲染状态时，将会应用到所有的*Pass*。如果我们不想这样（例如在双面渲染中，我们希望在第一个*Pass*中剔除正面来对背面进行渲染，在第二个*Pass*中剔除背面来对正面进行渲染），可以在*Pass*语义块中单独进行上面的设置。

• **SubShader**的标签

**SubShader**的标签（**Tags**）是一个键值对（**Key/Value Pair**），它的键和值都是字符串类型。这些键值对是**SubShader**和渲染引擎之间的沟通桥梁。它们用来告诉Unity的渲染引擎：我希望怎样以及何时渲染这个对象。

标签的结构如下：

```
Tags { "TagName1" = "Value1" "TagName2" = "Value2" }
```

**SubShader**的标签块支持的标签类型如表3.3所示。

表3.3      **SubShader**的标签类型

标 签 类 型	说 明	例 子
Queue	控制渲染顺序，指定该物体属于哪一个渲染队列，通过这种方式可以保证所有的透明物体可以在所有不透明物体后面被渲染（详见第8章），我们也可以自定义使用的渲染队列来控制物体的渲染顺序	Tags { "Queue" = "Transparent" }
RenderType	对着色器进行分类，例如这是一个不透明的着色器，或是一个透明的着色器等。这可以被用于着色器替换（Shader Replacement）功能	Tags { "RenderType" = "Opaque" }

标 签 类 型	说 明	例 子
DisableBatching	一些 <i>SubShader</i> 在使用Unity的批处理功能时会出现问题，例如使用了模型空间下的坐标进行顶点动画（详见11.3节）。这时可以通过该标签来直接指明是否对该 <i>SubShader</i> 使用批处理	Tags { "DisableBatching" = "True" }
ForceNoShadowCasting	控制使用该 <i>SubShader</i> 的物体是否会投射阴影（详见8.4节）	Tags { "ForceNoShadowCasting" = "True" }
IgnoreProjector	如果该标签值为“True”，那么使用该 <i>SubShader</i> 的物体将不会受Projector的影响。通常用于半透明物体	Tags { "IgnoreProjector" = "True" }
CanUseSpriteAtlas	当该 <i>SubShader</i> 是用于精灵（sprites）时，将该标签设为“False”	Tags { "CanUseSpriteAtlas" = "False" }
PreviewType	指明材质面板将如何预览该材质。默认情况下，材质将显示为一个球形，我们可以通过把该标签的值设为“Plane”“SkyBox”来改变预览类型	Tags { "PreviewType" = "Plane" }

具体的标签设置我们会在本书后面的章节中讲到。

需要注意的是，上述标签仅可以在`SubShader`中声明，而不可在`Pass`块中声明。`Pass`块虽然也可以定义标签，但这些标签是不同于`SubShader`的标签类型。这是我们下面将要讲到的。

- ***Pass***语义块

`Pass`语义块包含的语义如下：

```
Pass {  
    [Name]  
    [Tags]  
    [RenderSetup]  
    // Other code  
}
```

首先，我们可以在`Pass`中定义该`Pass`的名称，例如：

```
Name "MyPassName"
```

通过这个名称，我们可以使用ShaderLab的`UsePass`命令来直接使用其他Unity Shader中的`Pass`。例如：

```
UsePass "MyShader/MYPASSNAME"
```

这样可以提高代码的复用性。需要注意的是，由于Unity内部会把所有`Pass`的名称转换成大写字母的表示，因此，在使用`UsePass`命令时必须使用大写形式的名字。

其次，我们可以对`Pass`设置渲染状态。`SubShader`的状态设置同样适用于`Pass`。除了上面提到的状态设置外，在`Pass`中我们还可以使用固定管线的着色器（详见3.4.3节）命令。



*Pass*同样可以设置标签，但它的标签不同于*SubShader*的标签。这些标签也是用于告诉渲染引擎我们希望怎样来渲染该物体。表3.4给出了*Pass*中使用的标签类型。

表3.4 *Pass*的标签类型

标 签 类 型	说 明	例 子
LightMode	定义该 <i>Pass</i> 在Unity的渲染流水线中的角色	Tags { "LightMode" = "ForwardBase" }
RequireOptions	用于指定当满足某些条件时才渲染该 <i>Pass</i> ，它的值是一个由空格分隔的字符串。目前，Unity支持的选项有：SoftVegetation。在后面的版本中，可能会增加更多的选项	Tags { "RequireOptions" = "SoftVegetation" }

除了上面普通的*Pass*定义外，Unity Shader还支持一些特殊的*Pass*，以便进行代码复用或实现更复杂的效果。

- **UsePass**: 如我们之前提到的一样，可以使用该命令来复用其他Unity Shader中的*Pass*;
- **GrabPass**: 该*Pass*负责抓取屏幕并将结果存储在一张纹理中，以用于后续的*Pass*处理（详见10.2.2节）。

如果读者对上述出现的某些定义和名词无法理解，也不要担心。在本书后面的章节中，我们会对这些内容进行更加深入的讲解。

### 3.3.4 留一条后路: *Fallback*

紧跟在各个*SubShader*语义块后面的，可以是一个*Fallback*指令。它用于告诉Unity，“如果上面所有的*SubShader*在这块显卡上都不能运行，那么就使用这个最低级的*Shader*吧！”

它的语义如下：

```
Fallback "name"  
// 或者  
Fallback Off
```

如上所述，我们可以通过一个字符串来告诉Unity这个“最低级的Unity Shader”是谁。我们也可以任性地关闭*Fallback*功能，但一旦你这么做，你的意思大概就是：“如果一块显卡跑不了上面所有的*SubShader*，那就不要管它了！”

下面给出了一个使用*Fallback*语句的例子：

```
Fallback "VertexLit"
```

事实上，*Fallback*还会影响阴影的投射。在渲染阴影纹理时，Unity会在每个Unity Shader中寻找一个阴影投射的Pass。通常情况下，我们不需要自己专门实现一个Pass，这是因为*Fallback*使用的内置Shader中包含了这样一个通用的Pass。因此，为每个Unity Shader正确设置*Fallback*是非常重要的。更多关于Unity中阴影的实现，可以参见9.4节。

### 3.3.5 ShaderLab还有其他的语义吗

除了上述的语义，还有一些不常用到的语义。例如，如果我们不满足于Unity内置的属性类型，想要自定义材质面板的编辑界面，就可以使用*CustomEditor*语义来扩展编辑界面。我们还可以使用*Category*语义来对Unity Shader中的命令进行分组。由于这些命令很少用到，本书将不再进行深入的讲解。

## 3.4 Unity Shader的形式

在上面，我们讲了Unity Shader文件的结构以及ShaderLab的语法。尽管Unity Shader可以做的事情非常多（例如设置渲染状态等），但其最重要的任务还是指定各种着色器所需的代码。这些着色器代码可以写在*SubShader*语义块中（表面着色器的做法），也可以写在*Pass*语义块中（顶点/片元着色器和固定函数着色器的做法）。

在Unity中，我们可以使用下面3种形式来编写Unity Shader。而不管使用哪种形式，真正意义上的Shader代码都需要包含在ShaderLab语义块中，如下所示：

```
Shader "MyShader" {
    Properties {
        // 所需的各种属性
    }
    SubShader {
        // 真正意义上的Shader代码会出现在这里
        // 表面着色器 (Surface Shader) 或者
        // 顶点/片元着色器 (Vertex/Fragment Shader) 或者
        // 固定函数着色器 (Fixed Function Shader)
    }
    SubShader {
        // 和上一个SubShader类似
    }
}
```

### 3.4.1 Unity的宠儿：表面着色器

表面着色器（**Surface Shader**）是Unity自己创造的一种着色器代码类型。它需要的代码量很少，Unity在背后做了很多工作，但渲染的代价比较大。它在本质上和下面要讲到的顶点/片元着色器是一样的。也就是说，当给Unity提供一个表面着色器的时候，它在背后仍旧把它转换成对应的顶点/片元着色器。我们可以理解成，表面着色器是Unity对顶点/片元着色器的更高一层的抽象。它存在的价值在于，Unity为我们处理了很多光照细节，使得我们不需要再操心这些“烦人的事情”。

一个非常简单的表面着色器示例代码如下：

```
Shader "Custom/Simple Surface Shader" {
    SubShader {
        Tags { "RenderType" = "Opaque" }
        CGPROGRAM
        #pragma surface surf Lambert
        struct Input {
            float4 color : COLOR;
        };
        void surf (Input IN, inout SurfaceOutput o) {
            o.Albedo = 1;
        }
        ENDCG
    }
    Fallback "Diffuse"
}
```

从上述程序中可以看出，表面着色器被定义在**SubShader**语义块（而非**Pass**语义块）中的**CGPROGRAM**和**ENDCG**之间。原因是，表面着色器不需要开发者关心使用多少个**Pass**、每个**Pass**如何渲染等问题，Unity会在背后为我们做好这些事情。我们要做的只是告诉它：“嘿，使用这些纹理去填充颜色，使用这个法线纹理去填充法线，使用**Lambert**光照模型，其他的不要来烦我！”。

*CGPROGRAM*和*ENDCG*之间的代码是使用Cg/HLSL编写的，也就是说，我们需要把Cg/HLSL语言嵌套在ShaderLab语言中。值得注意的是，这里的Cg/HLSL是Unity经封装后提供的，它的语法和标准的Cg/HLSL语法几乎一样，但还是有细微的不同，例如有些原生的函数和用法Unity并没有提供支持。

### 3.4.2 最聪明的孩子：顶点/片元着色器

在Unity中我们可以使用Cg/HLSL语言来编写顶点/片元着色器（**Vertex/Fragment Shader**）。它们更加复杂，但灵活性也更高。

一个非常简单的顶点/片元着色器示例代码如下：

```
Shader "Custom/Simple VertexFragment Shader" {
    SubShader {
        Pass {
            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            float4 vert(float4 v : POSITION) : SV_POSITION {
                return mul (UNITY_MATRIX_MVP, v);
            }

            fixed4 frag() : SV_Target {
                return fixed4(1.0,0.0,0.0,1.0);
            }

            ENDCG
        }
    }
}
```

和表面着色器类似，顶点/片元着色器的代码也需要定义在*CGPROGRAM*和*ENDCG*之间，但不同的是，顶点/片元着色器是写在*Pass*语义块内，而非*SubShader*内的。原因是，我们需要自己定义每个

Pass需要使用的Shader代码。虽然我们可能需要编写更多的代码，但带来的好处是灵活性很高。更重要的是，我们可以控制渲染的实现细节。同样，这里的CGPROGRAM和ENDCG之间的代码也是使用Cg/HLSL编写的。

### 3.4.3 被抛弃的角落：固定函数着色器

上面两种Unity Shader形式都使用了可编程管线。而对于一些较旧的设备（其GPU仅支持DirectX 7.0、OpenGL 1.5或OpenGL ES 1.1），例如iPhone 3，它们不支持可编程管线着色器，因此，这时候我们就需要使用固定函数着色器（**Fixed Function Shader**）来完成渲染。这些着色器往往只可以完成一些非常简单的效果。

一个非常简单的固定函数着色器示例代码如下：

```
Shader "Tutorial/Basic" {
    Properties {
        _Color ("Main Color", Color) = (1,0.5,0.5,1)
    }
    SubShader {
        Pass {
            Material {
                Diffuse [_Color]
            }
            Lighting On
        }
    }
}
```

可以看出，固定函数着色器的代码被定义在`Pass`语义块中，这些代码相当于`Pass`中的一些渲染设置，正如我们之前在3.3.3节中提到的一样。



对于固定函数着色器来说，我们需要完全使用ShaderLab的语法（即使用ShaderLab的渲染设置命令）来编写，而非使用Cg/HLSL。

由于现在绝大多数GPU都支持可编程的渲染管线，这种固定管线的编程方式已经逐渐被抛弃。实际上，在Unity 5.2中，所有固定函数着色器都会在背后被Unity编译成对应的顶点/片元着色器，因此真正意义上的固定函数着色器已经不存在了。

### 3.4.4 选择哪种Unity Shader形式

那么，我们究竟选择哪一种来进行Unity Shader的编写呢？这里给出了一些建议。

- 除非你有非常明确的需求必须要使用固定函数着色器，例如需要在非常旧的设备上运行你的游戏（这些设备非常少见），否则请使用可编程管线的着色器，即表面着色器或顶点/片元着色器。
- 如果你想和各种光源打交道，你可能更喜欢使用表面着色器，但需要小心它在移动平台的性能表现。
- 如果你需要使用的光照数目非常少，例如只有一个平行光，那么使用顶点/片元着色器是一个更好的选择。
- 最重要的是，如果你有很多自定义的渲染效果，那么请选择顶点/片元着色器。

## 3.5 本书使用的Unity Shader形式

本书的目的不仅在于教给读者如何使用Unity Shader，更重要的是想要让读者掌握渲染背后的原理。仅仅了解高层抽象虽然可能会暂时

使工作简化，但从长久来看“知其然而不知其所以然”所带来的影响更加深远。

因此，在本书接下来的内容中，我们将着重使用顶点/片元着色器来进行Unity Shader的编写。对于表面着色器来说，我们会在本书的第17章中进行剖析，读者可以在那里找到更多的学习内容。

## 3.6 答疑解惑

尽管在之前的内容中涵盖了很多基础内容，这里仍给出一些初学者常见的困惑之处，并给予说明和解释。

### 3.6.1 Unity Shader != 真正的Shader

需要读者注意的是，Unity Shader并不等同于第2章中所讲的Shader，尽管Unity Shader翻译过来就是Unity着色器。在Unity里，Unity Shader实际上指的就是一个ShaderLab文件——硬盘上以.shader作为文件后缀的一种文件。

在Unity Shader（或者说是ShaderLab文件）里，我们可以做的事情远多于一个传统意义上的Shader。

- 在传统的Shader中，我们仅可以编写特定类型的Shader，例如顶点着色器、片元着色器等。而在Unity Shader中，我们可以在同一个文件里同时包含需要的顶点着色器和片元着色器代码。
- 在传统的Shader中，我们无法设置一些渲染设置，例如是否开启混合、深度测试等，这些是开发者在另外的代码中自行设置的。而

在Unity Shader中，我们通过一行特定的指令就可以完成这些设置。

- 在传统的Shader中，我们需要编写冗长的代码来设置着色器的输入和输出，要小心地处理这些输入输出的位置对应关系等。而在Unity Shader中，我们只需要在特定语句块中声明一些属性，就可以依靠材质来方便地改变这些属性。而且对于模型自带的属性（如顶点位置、纹理坐标、法线等），Unity Shader也提供了直接访问的方法，不需要开发者自行编码来传给着色器。

当然，Unity Shader除了上述这些优点外，也有一些缺点。由于Unity Shader的高度封装性，我们可以编写的Shader类型和语法都被限制了。对于一些类型的Shader，例如曲面细分着色器（Tessellation Shader）、几何着色器（Geometry Shader）等，Unity的支持就相对差一些。例如，Unity 4.x仅在DirectX 11平台下提供曲面细分着色器、几何着色器的相关功能，而对于OpenGL平台则没有这些支持。除此之外，一些高级的Shader语法Unity Shader也不支持。

可以说，Unity Shader提供了一种让开发者同时控制渲染流水线中多个阶段的一种方式，不仅仅是提供Shader代码。作为开发者而言，我们绝大部分时候只需要和Unity Shader打交道，而不需要关心渲染引擎底层的实现细节。

### 3.6.2 Unity Shader和Cg/HLSL之间的关系

正如我们之前所讲，Unity Shader是用ShaderLab语言编写的，但对于表面着色器和顶点/片元着色器，我们可以在ShaderLab内部嵌套Cg/HLSL语言来编写这些着色器代码。这些Cg/HLSL代码是嵌套在

*CGPROGRAM*和*ENDCG*之间的，正如我们之前看到的示例代码一样。由于Cg和DX9风格的HLSL从写法上来说几乎是同一种语言，因此在Unity里Cg和HLSL是等价的。我们可以说，Cg/HLSL代码是区别于ShaderLab的另一个世界。

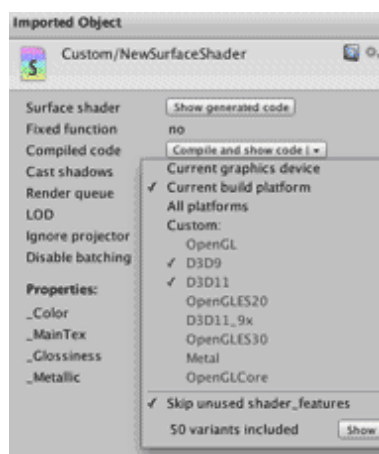
通常，Cg的代码片段是位于*Pass*语义块内部的，如下所示：

```
Pass {  
    // Pass的标签和状态设置  
  
    CGPROGRAM  
    // 编译指令，例如：  
    #pragma vertex vert  
    #pragma fragment frag  
  
    // Cg代码  
  
    ENDCG  
    // 其他一些设置  
}
```

读者可能会有疑问：“之前不是说在表面着色器中，Cg/HLSL代码是写在*SubShader*语义块内吗？而不是*Pass*块内。”的确，在表面着色器中，Cg/HLSL代码是写在*SubShader*语义块内，但是读者应该还记得，表面着色器在本质上就是顶点/片元着色器，它们看起来很不像是因为表面着色器是Unity在顶点/片元着色器上层为开发者提供的一层抽象封装，但在背后，Unity还是会把它转化成一个包含多*Pass*的顶点/片元着色器。我们可以在Unity Shader的导入设置面板中单击*Show generated code*按钮来查看生成的真正的顶点/片元着色器代码。可以说，从本质上来讲，Unity Shader只有两种形式：顶点/片元着色器和固定函数着色器（在Unity 5.2以后的版本中，固定函数着色器也会在背后被转化成顶点/片元着色器，因此从本质上来说Unity中只存在顶点/片元着色器）。

在提供给编程人员这些便利的背后，Unity编辑器会把这些Cg片段编译成低级语言，如汇编语言等。通常，Unity会自动把这些Cg片段编译到所有相关平台（这里的平台是指不同的渲染平台，例如Direct3D 9、OpenGL、Direct3D 11、OpenGL ES等）上。这些编译过程比较复杂，Unity会使用不同的编译器来把Cg转换成对应平台的代码。这样就不会在切换平台时再重新编译，而且如果代码在某些平台上发生错误就可以立刻得到错误信息。

正如在3.1.3节中看到的一样，我们可以在Unity Shader的导入设置面板上查看这些编译后的代码，查看这些代码有助于进行Debug或优化等，如图3.9所示。



▲ 图3.9 在Unity Shader的导入设置面板中可以通过*Compile and show code*按钮来查看Unity对CG片段编译后的代码。通过单击*Compile and show code*按钮右端的倒三角可以打开下拉菜单，在这个下拉菜单中可以选择编译的平台种类，如只为当前的显卡设备编译特定的汇编代码，或为所有的平台编译汇编代码，我们也可以自定义选择编译到哪些平台上

但当发布游戏的时候，游戏数据文件中只包含目标平台需要的编译代码，而那些在目标平台上不需要的代码部分就会被移除。例如，当发布到Mac OS X平台上时，DirectX对应的代码部分就会被移除。

### 3.6.3 我可以使用GLSL来写吗

当然可以。如果你坚持说：“我就是不想用Cg/HLSL来写！就是要使用GLSL来写！”，但是这意味着你可以发布的目标平台就只有Mac OS X、OpenGL ES 2.0或者Linux，而对于PC、Xbox 360这样的仅支持DirectX的平台来说，你就放弃它们了。

建立在你坚持要用GLSL来写Unity Shader的意愿下，你可以怎么写呢？和Cg/HLSL需要嵌套在*CGPROGRAM*和*ENDCG*之间类似，GLSL的代码需要嵌套在*GLSLPROGRAM*和*ENDGLSL*之间。

更多关于如何在Unity Shader中写GLSL代码的内容可以在Unity官方手册的**GLSL Shader Programs**一文

（<http://docs.unity3d.com/Manual/SL-GLSLShaderPrograms.html>）中找到。

## 3.7 扩展阅读

Unity官网上关于Unity Shader方面的文档正在不断补充中，由于Unity封装了很多功能和细节，因此，如果读者在使用Unity Shader的过程中遇到了问题可以去到官方文档（<http://docs.unity3d.com/Manual/SL-Reference.html>）中查看。除此之外，Unity也提供了一些简单的着色器编写教程（<http://docs.unity3d.com/Manual/ShaderTut1.html>，<http://docs.unity3d.com/Manual/ShaderTut2.html>）。由于在Unity Shader中，绝大多数可编程管线的着色器代码是使用Cg语言编写的，读者可以在NVIDIA提供的Cg文档（<http://http.developer.nvidia.com/Cg/>）中找到更多的内容。NVIDIA同样提供了一个系列教程



([http://http.developer.nvidia.com/CgTutorial/cgtutorial chapter01.html](http://http.developer.nvidia.com/CgTutorial/cgtutorial%20chapter01.html))

来帮助初学者掌握Cg的基本语法。

## 第4章 学习Shader所需的数学基础

不懂数学者不得入内。

——古希腊柏拉图学院门口的碑文

计算机图形学之所以深奥难懂，很大原因是在于它是建立在虚拟世界上的数学模型。数学渗透到图形学的方方面面，当然也包括Shader。在学习Shader的过程中，我们最常使用的就是矢量和矩阵（即数学的分支之一——线性代数）。

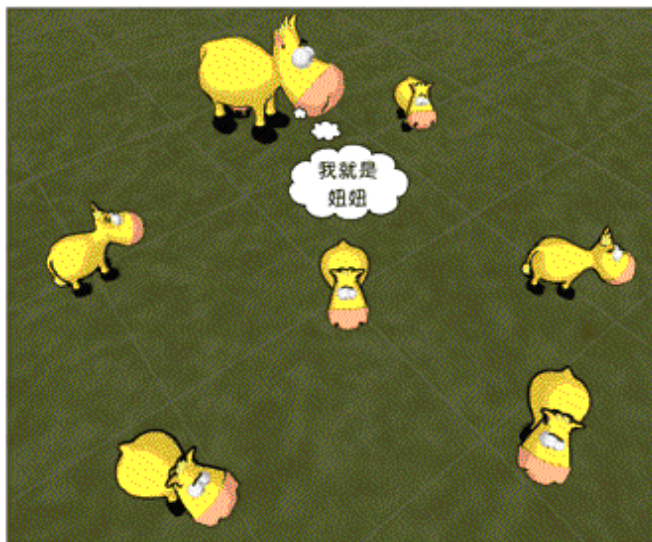
很多读者认为图形学中的数学复杂难懂。的确，一些数学模型在初学者看来晦涩难懂。但很多情况下，我们需要打交道的只是一些基础的数学运算，而只要掌握了这些内容，就会发现很多事情可以迎刃而解。我们在研究和学习他人编写的Shader代码时，也不再会疑问：“他为什么要这么写”，而是“哦，这里就是使用矩阵进行了一个变换而已。”

为了让读者能够参与到计算中来，而不是填鸭式地阅读，在一些小节最后我们会给出一些练习题。练习题的答案会在本章最后给出（不要偷看答案！）。需要注意的是，这些练习题并不是可有可无的，我们并非想利用题海战术来让读者掌握这些数学运算，而是想利用这些练习题来阐述一些容易出错或实践中常见的问题。通过这些练习题，读者可以对本节内容有更加深刻的理解。

那么，拿起笔来，让我们一起走进数学的世界吧！

## 4.1 背景：农场游戏

为了让读者更加理解数学计算的几何意义，我们先来假定一个场景。现在，假设我们正在开发一款卡通风格的农场游戏。在这个游戏里，玩家可以在农场里养很多可爱的奶牛。与普通农场游戏不同的是，我们的主角不是玩家，而是一头牛——妞妞，如图4.1所示。妞妞不仅长得壮，而且它对很多事情都充满了好奇心。



▲图4.1 我们的农场游戏。我们的主角妞妞是一头长得最壮、好奇心很强的奶牛

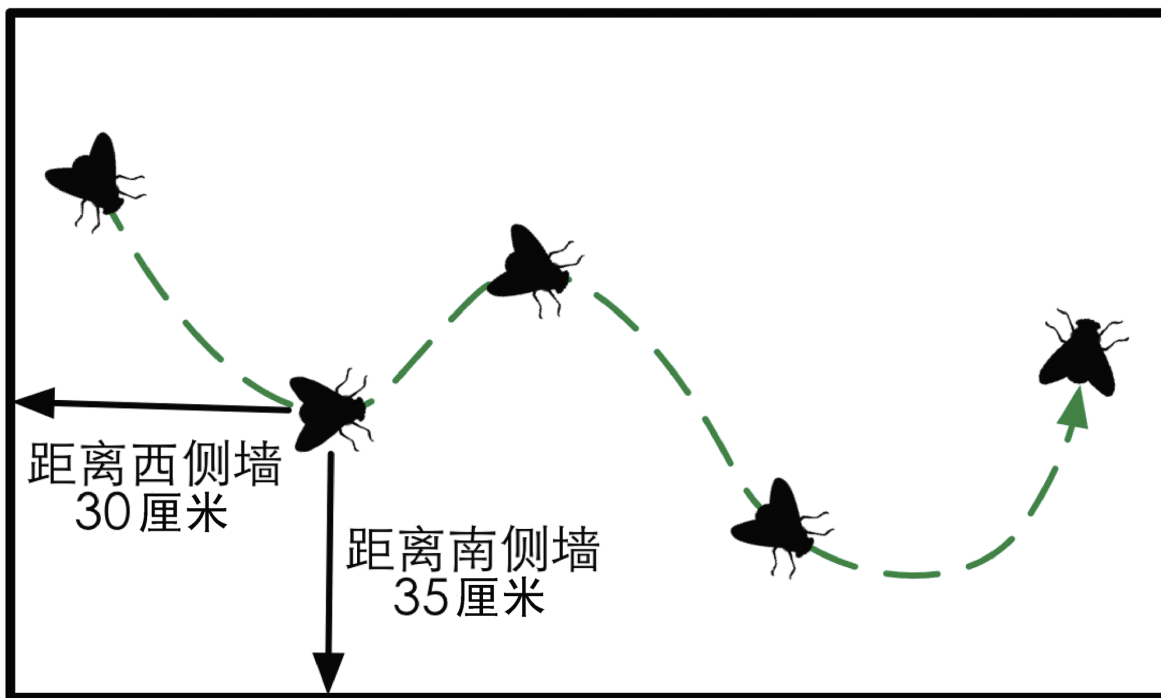
读者：为什么游戏主角不是玩家呢？我们：因为我们的策划就是这么任性。

在故事的一开始，农场世界是没有数学概念的。通过下面的学习，我们会见证数学给这个世界带来了怎样翻天覆地的变化。

## 4.2 笛卡儿坐标系

在游戏制作中，我们使用数学绝大部分都是为了计算位置、距离和角度等变量。而这些计算大部分都是在**笛卡儿坐标系（Cartesian Coordinate System）**下进行的。这个名字来源于法国伟大的哲学家、物理学家、心理学家、数学家笛卡儿（René Descartes）。

那么，我们为什么需要笛卡儿坐标系呢？有这样一个传说，讲述了笛卡儿提出笛卡儿坐标系的由来。笛卡儿从小体弱多病，所以他所在的寄宿学校的老师允许他可以一直留在床上直到中午。在笛卡儿的一生中，他每天的上午时光几乎都是在床上度过的。笛卡儿并没有把这段时间用在睡懒觉上，而是思考了很多关于数学和哲学上的问题。有一天，笛卡儿发现一只苍蝇在天花板上爬来爬去，他观察了很长一段时间。笛卡儿想：我要如何来描述这只苍蝇的运动轨迹呢？最后，笛卡儿意识到，他可以使用这只苍蝇距离房间内不同墙面的位置来描述，如图4.2所示。他从床上起身，写下了他的发现。然后，他试图描述一些点的位置，正如他要描述苍蝇的位置一样。最后，笛卡儿就发明了这个坐标平面。而这个坐标平面后来逐渐发展，就形成了坐标系系统。人们为了纪念笛卡儿的工作，就用他的名字来给这种坐标系进行命名。



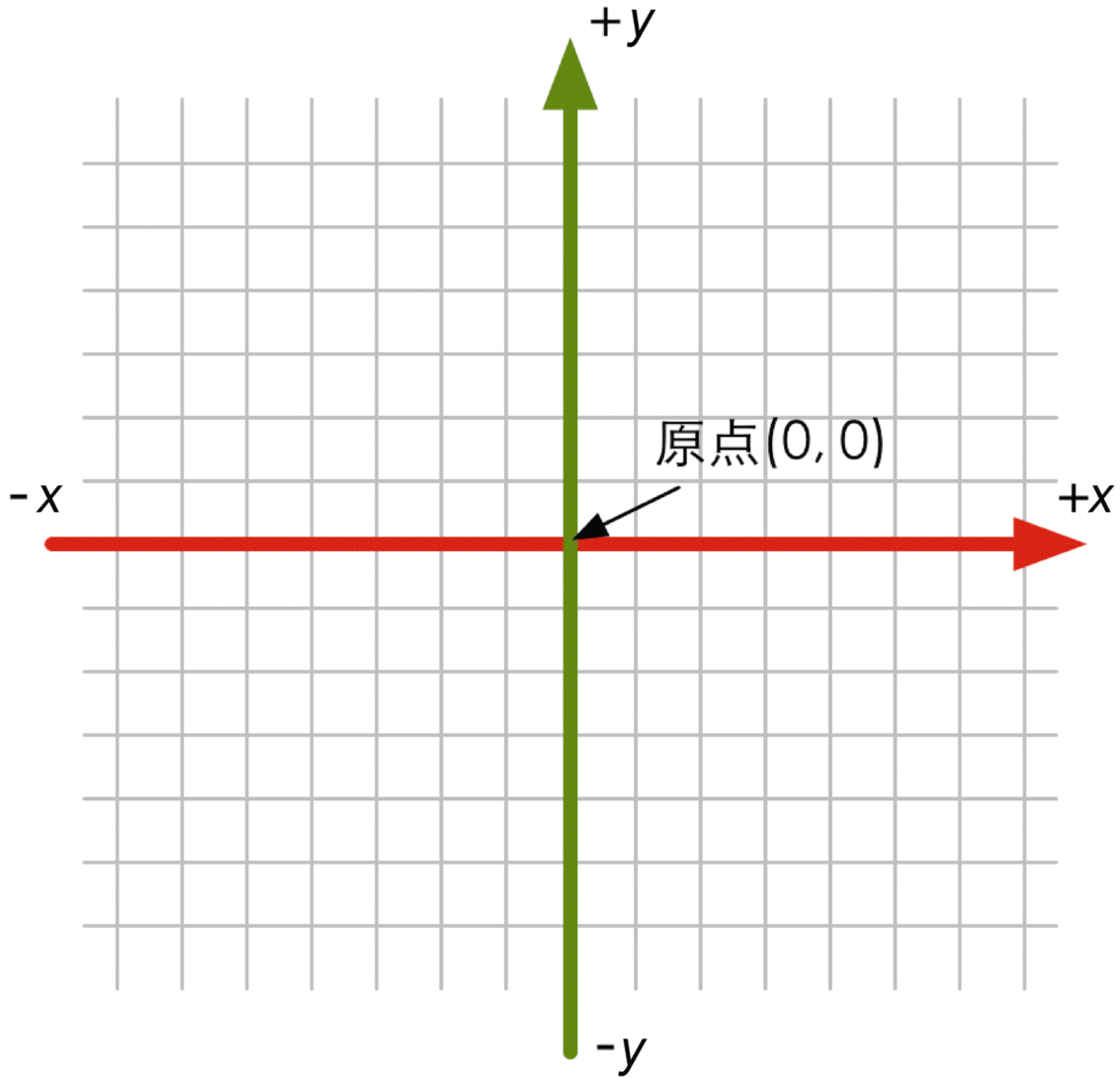
▲图4.2 传说，笛卡儿坐标系来源于笛卡儿对天花板上一只苍蝇的运动轨迹的观察。笛卡儿发现，可以使用苍蝇距不同墙面的距离来描述它的当前位置

当然，上面传说的可靠性无从验证。一些较真儿的读者就不用急着向本书勘误邮箱中发邮件说：“嘿，你简直是胡说！”不过，读者可以从这个传说中发现，笛卡儿坐标系和我们的生活是密切相关的。

### 4.2.1 二维笛卡儿坐标系

事实上，读者很可能一直在用二维笛卡儿坐标系，尽管你可能并没有听过笛卡儿这个名词。你还记得在《哈利波特与魔法石》电影中，哈利和罗恩大战奇洛教授的魔法棋盘吗？这里的国际象棋棋盘也可以理解成是一个二维的笛卡儿坐标系。

图4.3显示了一个二维笛卡儿坐标系。它是不是很像一个棋盘呢？



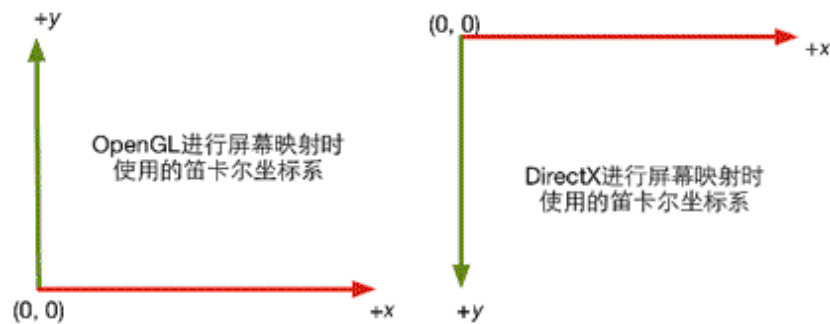
▲图4.3 一个二维笛卡尔坐标系

一个二维的笛卡尔坐标系包含了两个部分的信息：

- 一个特殊的位置，即原点，它是整个坐标系的中心；
- 两条过原点的互相垂直的矢量，即x轴和y轴。这些坐标轴也被称为是该坐标系的基矢量。



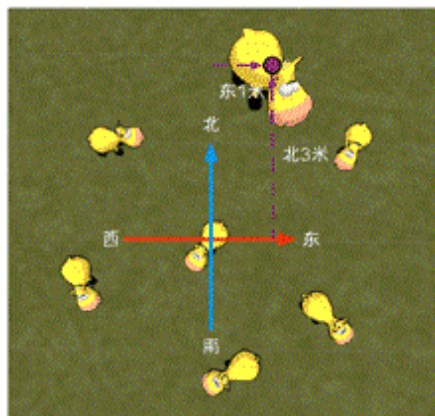
虽然在图4.3中x轴和y轴分别是水平和垂直方向的，但这并不是必须的。想象把上面的坐标系整体向左旋转30°。而且，虽然图中的x轴指向右、y轴指向上，但这也并不是必须的。例如，在2.3.4节屏幕映射中，OpenGL和DirectX使用了不同的二维笛卡尔坐标系，如图4.4所示。



▲ 图4.4 在屏幕映射时，OpenGL和DirectX使用了不同方向的二维笛卡尔坐标系

而有了这个坐标系我们就可以精确地定位一个点的位置。例如，如果说：“在（1, 2）的位置上画一个点。”那么相信读者肯定知道这个位置在哪里。

我们来看一下笛卡尔坐标系给奶牛农场带来了什么变化。在没有笛卡尔坐标系的时候，奶牛们根本没有明确的位置概念。如果一头奶牛问：“妞妞，你现在在哪里啊？”妞妞只能回答说“我在这里”或者“我在那里”这些模糊的词语。但那头奶牛永远不会知道妞妞的确切位置。而把笛卡尔坐标系引入到奶牛农场后，所有的一切都变得清晰起来。我们把奶牛农场的中心定义成坐标原点，而把地理方向中的东、北定义成坐标轴方向。现在，如果奶牛再问：“妞妞，你现在在哪里啊？”妞妞就可以回答说：“我在东1米、北3米的地方。”如图4.5所示。



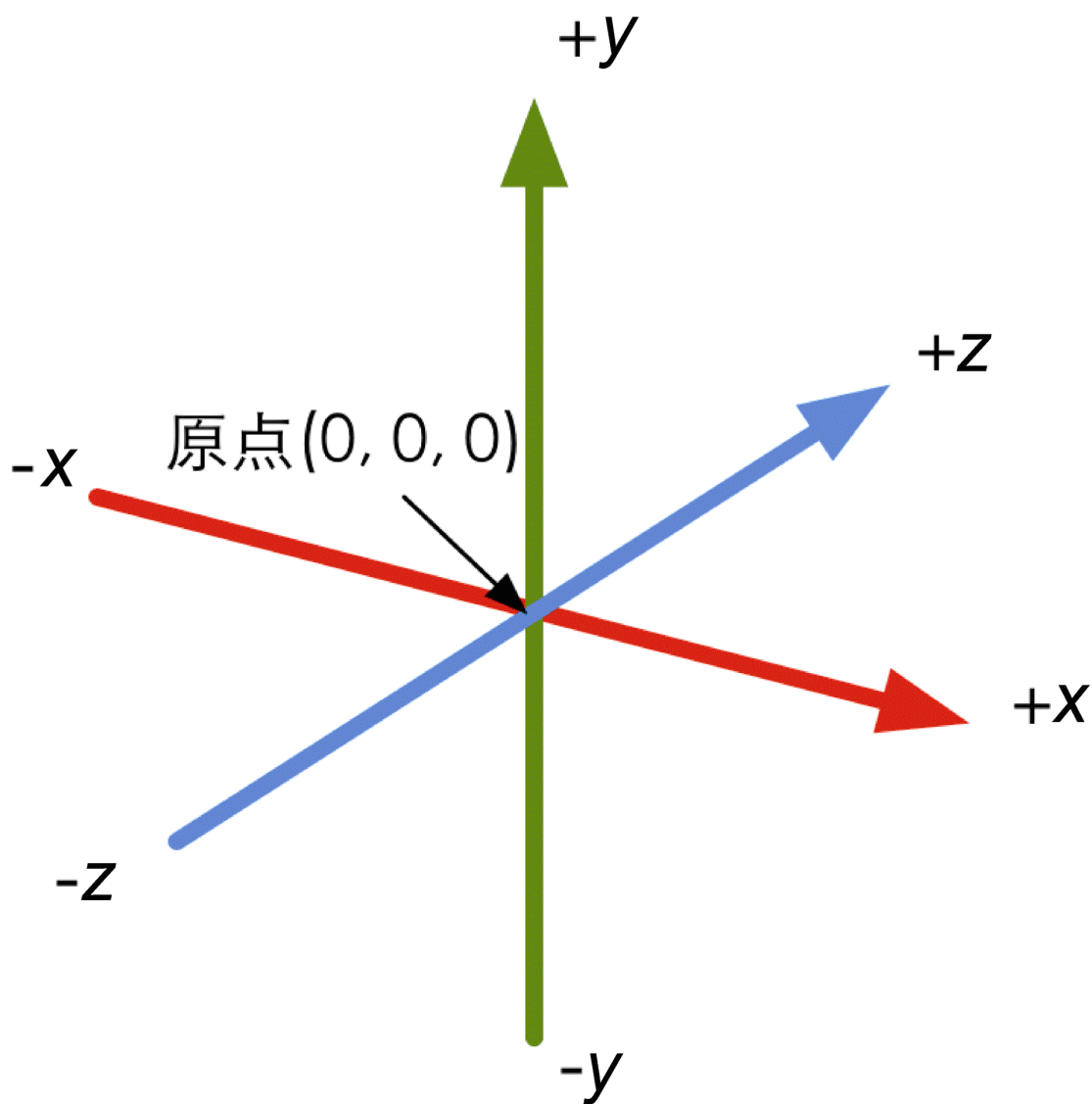
▲图4.5 笛卡儿坐标系可以让妞妞精确表述自己的位置

## 4.2.2 三维笛卡儿坐标系

在上面一节中，我们已经了解了二维笛卡儿坐标系。可以看出，二维笛卡儿坐标系实际上是比较简单的。那么，三维比二维只多了一个维度，是不是也就难了50%而已呢？

不幸的是，答案是否定的。三维笛卡儿坐标系相较于二维来说要复杂许多，但这并不意味着很难学会它。对人类来说，我们生活的世界就是三维的，因此对于理解更低维度的空间（一维和二维）是比较容易的。而对于同等维度的一些概念；理解起来难度就大一些；对于更高维度的空间（如四维空间），理解难度就更大了。

在三维笛卡儿坐标系中，我们需要定义3个坐标轴和一个原点。图4.6显示了一个三维笛卡儿坐标系。



▲图4.6 一个三维笛卡儿坐标系

这3个坐标轴也被称为是该坐标系的**基矢量 (basis vector)**。通常情况下，这3个坐标轴之间是互相垂直的，且长度为1，这样的基矢量被称为**标准正交基 (orthonormal basis)**，但这并不是必须的。例如，在一些坐标系中坐标轴之间互相垂直但长度不为1，这样的基矢量被称

为**正交基**（**orthogonal basis**）。如非特殊说明，本书默认情况下使用的坐标轴指的都是标准正交基。

读者：正交这个词是什么意思呢？

我们：正交可以理解成互相垂直的意思。在下面矩阵的内容中，我们还会看到正交矩阵的概念。

和二维笛卡儿坐标系类似，三维笛卡儿坐标系中的坐标轴方向也不是固定的，即不一定是像图4.6中那样的指向。但这种不同导致了两种不同种类的坐标系：**左手坐标系**（**left-handed coordinate space**）和**右手坐标系**（**right-handed coordinate space**）。

### 4.2.3 左手坐标系和右手坐标系

为什么在三维笛卡儿坐标系中要区分左手坐标系和右手坐标系，而二维中就没有这些烦人的事情呢？这是因为，在二维笛卡儿坐标系中， $x$ 轴和 $y$ 轴的指向虽然可能不同，就如我们在图4.4中看到的一样。但我们总可以通过一些旋转操作来使它们的坐标轴指向相同。以图4.4中OpenGL和DirectX使用的坐标系为例，为了把右侧的坐标轴指向转换到左侧那样的指向，我们可以首先对右侧的坐标系顺时针旋转 $180^\circ$ ，此时它的 $y$ 轴指向上，而 $x$ 轴指向左。然后，我们再把整个纸面水平翻转一下，就可以把 $x$ 轴翻转到指向右了，此时左右两侧的坐标轴指向就完全相同了。从这种意义上来说，所有的二维笛卡儿坐标系都是等价的。

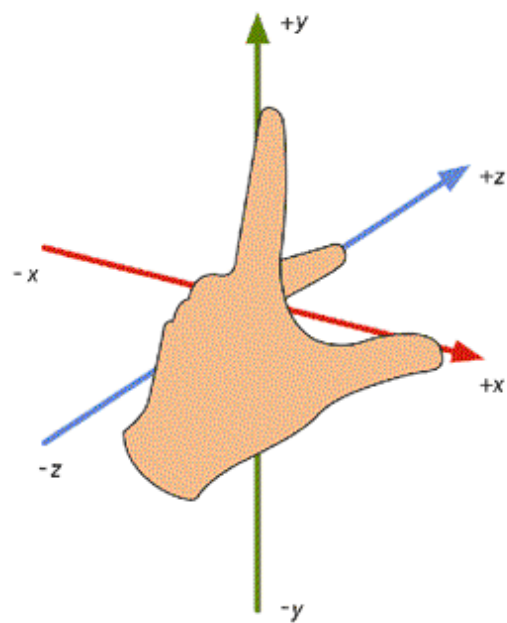
但对于三维笛卡儿坐标系，靠这种旋转有时并不能使两个不同朝向的坐标系重合。例如，在图4.6中， $+z$ 轴的方向指向纸面的内部，如

果有另一个三维笛卡儿坐标系，它的 $+z$ 轴是指向纸面外部， $x$ 轴和 $y$ 轴保持不变，那么我们可以通过旋转把这两个坐标轴重合在一起吗？答案是否定的。我们总可以让其中两个坐标轴的指向重合，但第三个坐标轴的指向总是相反的。

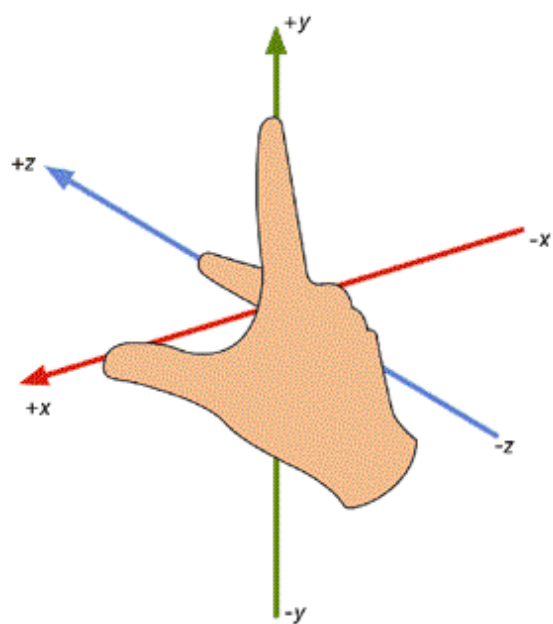
也就是说，三维笛卡儿坐标系并不都是等价的。因此，就出现了两种不同的三维坐标系：左手坐标系和右手坐标系。如果两个坐标系具有相同的**旋向性**（**handedness**），那么我们就可以通过旋转的方法来让它们的坐标轴指向重合。但是，如果它们具有不同的旋向性（例如坐标系**A**属于左手坐标系，而坐标系**B**属于右手坐标系），那么就无法达到重合的目的。

那么，为什么叫左手坐标系和右手坐标系呢？和手有什么关系？这是因为，我们可以利用我们的双手来判断一个坐标系的旋向性。请读者举起你的左手，用食指和大拇指摆出一个“**L**”的手势，并且让你的食指指向上，大拇指指向右。现在，伸出你的中指，不出意外的话它应该指向你的前方（如果你一定要展示自己骨骼惊奇的话我也没有办法）。恭喜你，你已经得到了一个左手坐标系了！你的大拇指、食指和中指分别对应了 $+x$ 、 $+y$ 和 $+z$ 轴的方向，如图4.7所示。

同样，读者可以通过右手来得到一个右手坐标系。举起你的右手，这次食指仍然指向上，中指指向前方，不同的是，大拇指将指向左侧，如图4.8所示。



▲ 图4.7 左手坐标系



▲ 图4.8 右手坐标系



正如我们之前所说，左手坐标系和右手坐标系之间无法通过旋转来同时使它们的3个坐标轴指向重合，如果你不信，你现在可以拿自己的双手来试验一下。

另外一个确定是左手还是右手坐标系的方法是，判断**前向（forward）**的方向。请读者坐直，向右伸直你的右手，此时右手方向就是x轴的正向，而你的头顶向上的方向就是y轴的正向。这时，如果你的正前方的方向是z轴的正向，那么你本身所在的坐标系就是一个左手坐标系；如果你的正前方的方向对应的是z轴的负向，那么这就是一个右手坐标系。

除了坐标轴朝向不同之外，左手坐标系和右手坐标系对于正向旋转的定义也不同，即在初高中物理中学到的**左手法则（left-hand rule）**和**右手法则（right-hand rule）**。假设现在空间中有一条直线，还有一个点，我们希望把这个点以该直线为旋转轴旋转某个角度，比如旋转 $30^\circ$ 。读者可以拿一支笔当成这个旋转轴，再拿自己的手当成这个需要旋转的点，可以发现，我们有两个旋转方向可以选择。那么，我们应该往哪个方向旋转呢？这意味着，我们需要在坐标系中定义一个旋转的正方向。在左手坐标系中，这个旋转正方向是由左手法则定义的，而在右手坐标系中则是由右手法则定义的。

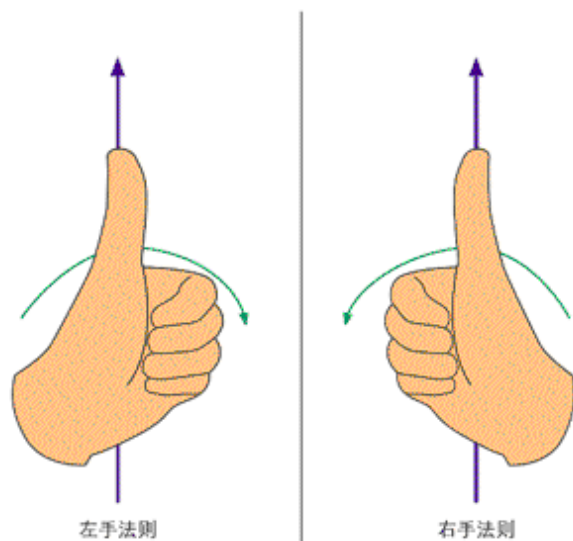
在左手坐标系中，我们可以这样来应用左手法则：还是举起你的左手，握拳，伸出大拇指让它指向旋转轴的正方向，那么旋转的正方向就是剩下4个手指的弯曲方向。在右手坐标系中，使用右手法则对旋转正方向的判断类似。如图4.9所示。

从图3.9中可以看出，在左手坐标系中，旋转正方向是顺时针的，而在右手坐标系中，旋转正方向是逆时针的。

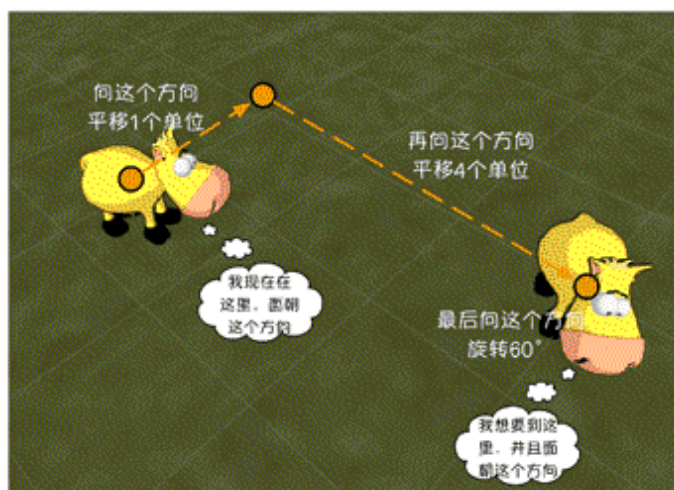
左右手坐标系之间是可以进行互相转换的。最简单的方法就是把其中一个轴反转，并保持其他两个轴不变。

对于开发者来说，使用左手坐标系还是右手坐标系都是可以的，它们之间并没有优劣之分。无论使用哪种坐标系，绝大多数情况下并不会影响底层的数学运算，而只是在映射到视觉上时会有差别（见练习题2）。这是因为，一个点或者旋转在空间内来说是绝对的。一些较真儿读者可能会看不惯“绝对”这个词：“你怎么能忽略相对论呢？这世上一切都是相对的！”这些读者请容我解释。这里所说的绝对是说，在我们所关心的最广阔的空间中，这些值是绝对的。例如我说，把你的书从桌子的左边移到右边，你不会对这个过程产生什么疑问，此时我们关心的整个空间就是桌子这个空间，而在这个空间中，书的运动是绝对的。但是，在数学的世界中，我们需要使用一种数学模型来精确地描述它们，这个模型就是坐标系。一旦有了坐标系，每个点的位置就不再是绝对的，而是相对于这个坐标系来说的。这种相对关系导致，即便从数学表示上来说两种表示方式完全一样，但从视觉上来说是不一样的。

我们可以在奶牛农场的例子中体会左手坐标系和右手坐标系的分别。我们假设，妞妞想要到一个新的地方，因为那里的草很美味。妞妞知道到达这个目标点的“绝对路径”是怎样的，如图4.10所示。

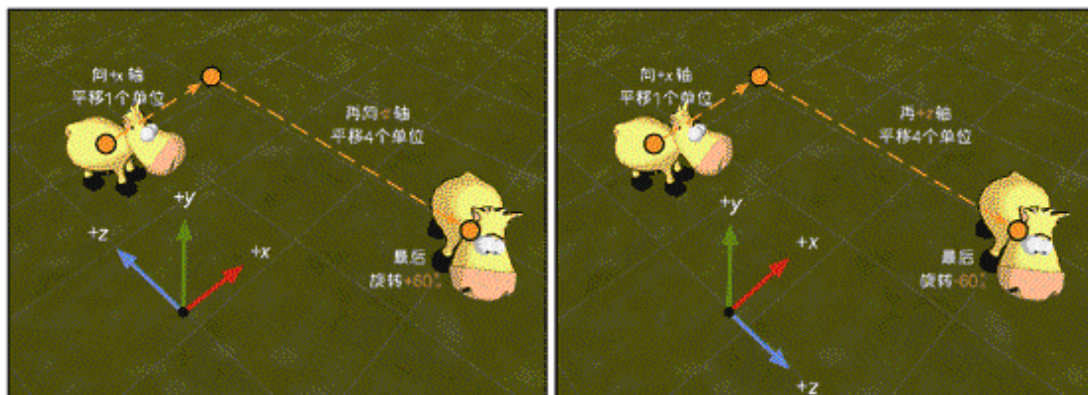


▲图4.9 用左手法则和右手法则来判断旋转正方向



▲图4.10 为了移动到新的位置，妞妞需要首先向某个方向平移1个单位，再向另一个方向平移4个单位，最后再向一个方向旋转 $60^\circ$

我们可以分别在一个左手坐标系和右手坐标系中描述这样一次运动，即使用数学表达式来描述它。我们会发现，在不同的坐标系中描述这样同一次运动是不一样的，如图4.11所示。



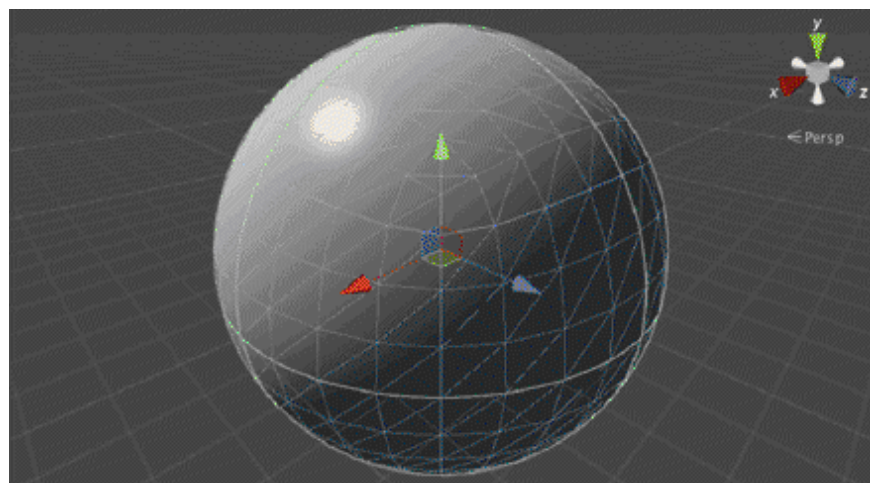
▲ 图4.11 左图和右图分别表示了左手坐标系和右手坐标系中描述妞妞这次运动的结果，得到的数学描述是不同的

在左手坐标系中，3个坐标轴的朝向如图4.11左图所示。妞妞首先向x轴正方向平移1个单位，然后再向z轴负方向移动4个单位，最后朝旋转的正方向旋转 $60^\circ$ 。而在右手坐标系中，+z轴的方向和左手坐标系中刚好相反，因此妞妞首先向x轴正方向平移1个单位（与左手坐标系中的移动一致），然后再向z轴正方向移动4个单位（与左手坐标系中的移动相反），最后朝旋转的负方向旋转 $60^\circ$ （与左手坐标系中的旋转相反）。

可以看出，为了达到同样的视觉效果（这里指把妞妞移动到视觉上的同一个位置），左右手坐标系在z轴上的移动以及旋转方向是不同的。如果使用相同的数学运算（指均向z轴某方向移动或均朝旋转正方向旋转等），那么得到的视觉效果就是不一样的。因此，如果我们需要从左手坐标系迁移到右手坐标系，并且保持视觉上的不变，就需要进行一些转换。读者可以参见本章最后的扩展阅读部分。

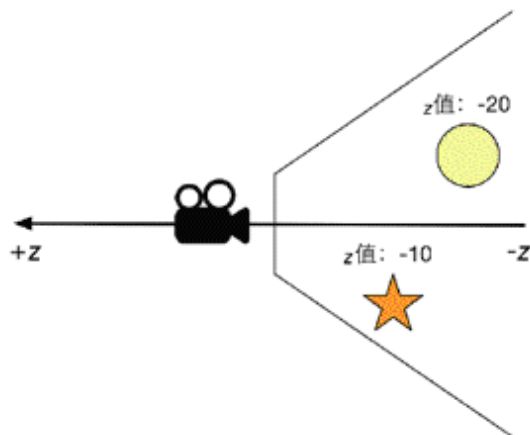
#### 4.2.4 Unity使用的坐标系

对于一个需要可视化虚拟的三维世界的应用（如Unity）来说，它的设计者就要进行一个选择。对于模型空间和世界空间（在4.6节中会具体讲解这两个空间是什么），Unity使用的是左手坐标系。这可以从Scene视图的坐标轴显示看出来，如图4.12所示。这意味着，在模型空间中，一个物体的右侧（right）、上侧（up）和前侧（forward）分别对应了x轴、y轴和z轴的正方向。



▲图4.12 在模型空间和世界空间中，Unity使用的是左手坐标系。图中，球的坐标轴显示了它在模型空间中的3个坐标轴（红色为x轴，绿色是y轴，蓝色是z轴）

但对于观察空间来说，Unity使用的是右手坐标系。观察空间，通俗来讲就是以摄像机为原点的坐标系。在这个坐标系中，摄像机的前向是z轴的负方向，这与在模型空间和世界空间中的定义相反。也就是说，z轴坐标的减少意味着场景深度的增加，如图4.13所示。



▲图4.13 在Unity中，观察空间使用的是右手坐标系，摄像机的前向是 $z$ 轴的负方向， $z$ 轴越小，物体的深度越大，离摄像机越远

关于Unity中使用的坐标系的旋向性，我们会在4.5.9节中详细地讲解。

### 4.2.5 练习题

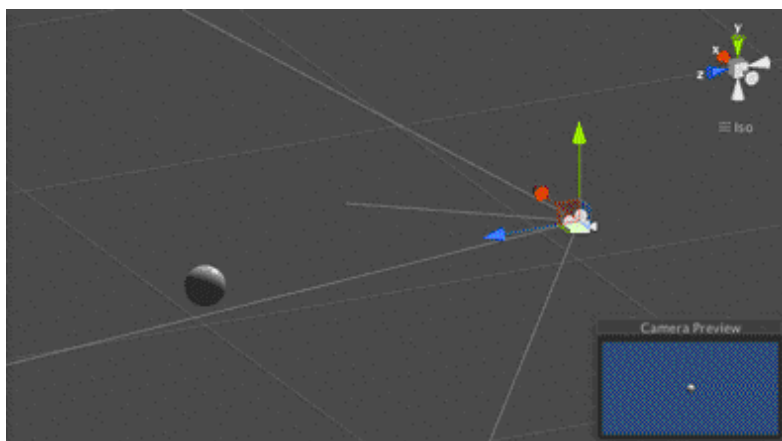
这是本书中第一次出现练习题的地方，希望你可以快速解决它们！

(1) 在非常流行的建模软件3ds Max中，默认的坐标轴方向是： $x$ 轴正方向指向右方， $y$ 轴正方向指向前方， $z$ 轴正方向指向上方。那么它是左手坐标系还是右手坐标系？

(2) 在左手坐标系中，有一点的坐标是  $(0, 0, 1)$ ，如果把该点绕 $y$ 轴正方向旋转 $+90^\circ$ ，旋转后的坐标是什么？如果是在右手坐标系中，同样有一点坐标为  $(0, 0, 1)$ ，把它绕 $y$ 轴正方向旋转 $+90^\circ$ ，旋转后的坐标是什么？



(3) 在Unity中，新建的场景中主摄像机的位置位于世界空间中的(0, 1, -10)位置。在不改变摄像机的任何设置（如保持Rotation为(0, 0, 0)，Scale为(1, 1, 1)）的情况下，在世界空间中的(0, 1, 0)位置新建一个球体，如图4.14所示。



▲图4.14 摄像机的位置是(0, 1, -10)，球体的位置是(0, 1, 0)

在摄像机的观察空间下，该球体的z值是多少？在摄像机的模型空间下，该球体的z值又是多少？

## 4.3 点和矢量

**点 (point)** 是 $n$ 维空间（游戏中主要使用二维和三维空间）中的一个位置，它没有大小、宽度这类概念。在笛卡尔坐标系中，我们可以使用2个或3个实数来表示一个点的坐标，如： $P=(P_x, P_y)$ ，表示二维空间的点， $P=(P_x, P_y, P_z)$ 表示三维空间中的点。

**矢量 (vector，也被称为向量)** 的定义则复杂一些。在数学家看来，矢量就是一串数字。你可能要问了，点的表达式不也是一串数字

吗？没错，但矢量存在的意义更多是为了和**标量**（**scalar**）区分开来。通常来讲，矢量是指 $n$ 维空间中一种包含了**模**（**magnitude**）和**方向**（**direction**）的**有向线段**，我们通常讲到的速度（**velocity**）就是一种典型的矢量。例如，这辆车的速度是向南80km/h（向南指明了矢量的方向，80km/h指明了矢量的模）。而标量只有模没有方向，生活中常常说到的距离（**distance**）就是一种标量。例如，我家离学校只有200米（200米就是一个标量）。

具体来讲。

- 矢量的模指的是这个矢量的长度。一个矢量的长度可以是任意的非负数。
- 矢量的方向则描述了这个矢量在空间中的指向。

矢量的表示方法和点类似。我们可以使用 $\mathbf{v}=(x,y)$ 来表示二维矢量，用 $\mathbf{v}=(x,y,z)$ 来表示三维矢量，用 $\mathbf{v}=(x,y,z,w)$ 来表示四维矢量。

为了方便阐述，我们对不同类型的变量在书写和印刷上使用不同的样式：

- 对于标量，我们使用小写字母来表示，如 $a$ ， $b$ ， $x$ ， $y$ ， $z$ ， $\theta$ ， $\alpha$ 等；
- 对于矢量，我们使用小写的粗体字母来表示，如 $\mathbf{a}$ ， $\mathbf{b}$ ， $\mathbf{u}$ ， $\mathbf{v}$ 等；
- 对于后面要学习的矩阵，我们使用大写的粗体字母来表示，如 $\mathbf{A}$ ， $\mathbf{B}$ ， $\mathbf{S}$ ， $\mathbf{M}$ ， $\mathbf{R}$ 等。

在图4.15中，一个矢量通常由一个箭头来表示。我们有时会讲到一个矢量的**头**（**head**）和**尾**（**tail**）。矢量的头指的是它的箭头所在的端

点处，而尾指的是另一个端点处，如图4.15所示。

那么一个矢量要放在哪里呢？从矢量的定义来看，它只有模和方向两个属性，并没有位置信息。这听起来很难理解，但实际上在生活中我们总是会这样的矢量打交道。例如，当我们讲到一个物体的速度时，可能会这样说“那个小偷正在以100km/h的速度向南逃窜”（快抓住他！），这里的“以100km/h的速度向南”就可以使用一个矢量来表示。通常，矢量被用于表示相对于某个点的**偏移（displacement）**，也就是说它是一个相对量。只要矢量的模和方向保持不变，无论放在哪里，都是同一个矢量。

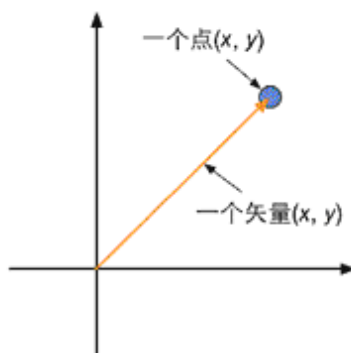
### 4.3.1 点和矢量的区别

回顾一下，点是一个没有大小之分的空间中的位置，而矢量是一个有模和方向但没有位置的量。从这里看，点和矢量具有不同的意义。但是，从表示方式上两者非常相似。

在上一节中我们提到，矢量通常用于描述偏移量，因此，它们可以用于描述相对位置，即相对于另一个点的位置，此时矢量的尾是一个位置，那么矢量的头就可以表示另一个位置了。而一个点可以用于指定空间中的一个位置（即相对于原点的位置）。如果我们把矢量的尾固定在坐标系原点，那么这个矢量的表示就和点的表示重合了。图4.16表示了两者的关系。



▲图4.15 一个二维向量以及它的头和尾



▲图4.16 点和矢量之间的关系

尽管上面的内容看起来显而易见，但区分点和矢量之间的不同是非常重要的，尽管它们在数学表达式上是一样的，都是一串数字而已。如果一定要给它们之间建立一个联系的话，我们可以认为，任何一个点都可以表示成一个从原点出发的矢量。为了明确点和矢量的区别，在本书后面的内容中，我们将用于表示方向的矢量称为方向矢量。

### 4.3.2 矢量运算

在下面的内容里，我们将给出一些最常见的矢量运算。幸运的是，这些运算大都很好理解。对于每种运算，我们会先给出数学上的描述，然后再给出几何意义上的解释。同样，为了让读者加深印象，

我们会在最后给出一些练习题。相信读完本节后，你一定可以快速地解决它们！

## 1. 矢量和标量乘法/除法

还记得吗？标量是只有模没有方向的量，虽然我们不能把矢量和标量进行相加/相减的运算（想象一下，你会把速度和距离相加吗），但可以对它们进行乘法运算，结果会得到一个不同长度且可能方向相反的新的矢量。

公式非常简单，我们只需要把矢量的每个分量和标量相乘即可：

$$k\mathbf{v}=(kv_x, kv_y, kv_z)$$

类似的，一个矢量也可以被一个非零的标量除。这等同于和这个标量的倒数相乘：

$$\frac{\mathbf{v}}{k} = \frac{(x, y, z)}{k} = \frac{1}{k}(x, y, z) = \left(\frac{x}{k}, \frac{y}{k}, \frac{z}{k}\right), k \neq 0$$

下面给出一些例子：

$$2(1, 2, 3) = (2, 4, 6)$$

$$-3.5(2, 0) = (-7, 0)$$

$$\frac{(1, 2, 3)}{2} = (0.5, 1, 1.5)$$

注意，对于乘法来说，矢量和标量的位置可以互换。但对于除法，只能是矢量被标量除，而不能是标量被矢量除，这是没有意义

的。

从几何意义上看，把一个矢量 $\mathbf{v}$ 和一个标量 $k$ 相乘，意味着对矢量 $\mathbf{v}$ 进行一个大小为 $|k|$ 的缩放。例如，如果想要把一个矢量放大两倍，就可以乘以2。当 $k < 0$ 时，矢量的方向也会取反。图4.17显示了这样的一些例子。

## 2. 矢量的加法和减法

我们可以对两个矢量进行相加或相减，其结果是一个相同维度的新矢量。

我们只需要把两个矢量的对应分量进行相加或相减即可。公式如下：

$$\mathbf{a} + \mathbf{b} = (a_x + b_x, a_y + b_y, a_z + b_z)$$

$$\mathbf{a} - \mathbf{b} = (a_x - b_x, a_y - b_y, a_z - b_z)$$

下面是一些例子：

$$(1, 2, 3) + (4, 5, 6) = (5, 7, 9)$$

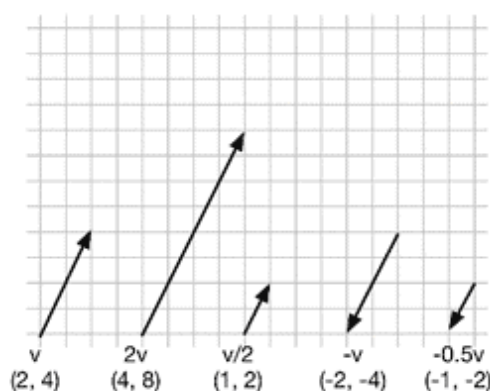
$$(5, 2, 7) - (3, 8, 4) = (2, -6, 3)$$

需要注意的是，一个矢量不可以和一个标量相加或相减，或者是和不同维度的矢量进行运算。

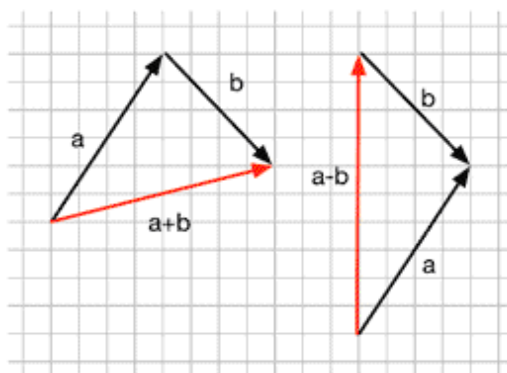
从几何意义上来看，对于加法，我们可以把矢量 $\mathbf{a}$ 的头连接到矢量 $\mathbf{b}$ 的尾，然后画一条从 $\mathbf{a}$ 的尾到 $\mathbf{b}$ 的头的矢量，来得到 $\mathbf{a}$ 和 $\mathbf{b}$ 相加后的矢



量。也就是说，如果我们从一个起点开始进行了一个位置偏移 $\mathbf{a}$ ，然后又进行一个位置偏移 $\mathbf{b}$ ，那么就等同于进行了一个 $\mathbf{a}+\mathbf{b}$ 的位置偏移。这被称为矢量加法的**三角形定则**（**triangle rule**）。矢量的减法是类似的。如图4.18所示。



▲ 图4.17 二维矢量和一些标量的乘法和除法



▲ 图4.18 二维矢量的加法和减法

读者需要时刻谨记，在图形学中矢量通常用于描述位置偏移（简称位移）。因此，我们可以利用矢量的加法和减法来计算一点相对于另一点的位移。

假设，空间内有两点 $a$ 和 $b$ 。还记得吗，我们可以用矢量 $\mathbf{a}$ 和 $\mathbf{b}$ 来表示它们相对于原点的位移。如果我们想要计算点 $b$ 相对于点 $a$ 的位移，就可以通过把 $\mathbf{b}$ 和 $\mathbf{a}$ 相减得到，如图4.19所示。

### 3. 矢量的模

正如我们之前讲到的一样，矢量是有模和方向的。矢量的模是一个标量，可以理解为是矢量在空间中的长度。它的表示符号通常是在矢量两旁分别加上一条垂直线（有的文献中会使用两条垂直线）。三维矢量的模的计算公式如下：

$$|\mathbf{v}| = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

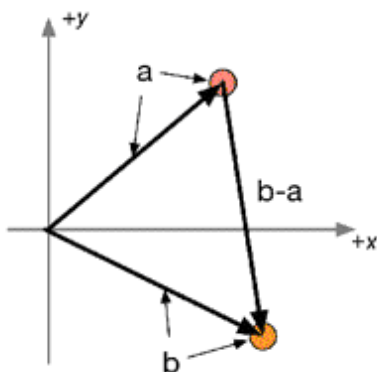
其他维度的矢量的模计算类似，都是对每个分量的平方相加后再开根号得到。

下面给出一些例子：

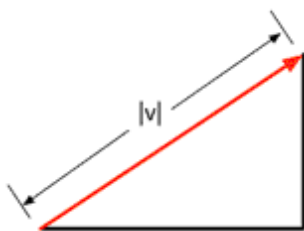
$$|(1, 2, 3)| = \sqrt{1^2 + 2^2 + 3^2} = \sqrt{1 + 4 + 9} = \sqrt{14} \approx 3.742$$

$$|(3, 4)| = \sqrt{3^2 + 4^2} = \sqrt{9 + 16} = \sqrt{25} = 5$$

我们可以从几何意义来理解上述公式。对于二维矢量来说，我们可以对任意矢量构建一个三角形，如图4.20所示。



▲图4.19 使用矢量减法来计算从点a到点b的位移



▲图4.20 矢量的模

从图4.20可以看出，对于二维矢量，其实就是使用了勾股定理，矢量的两个分量的绝对值对应了三角形两个直角边的长度，而斜边的长度就是矢量的模。

#### 4. 单位矢量

在很多情况下，我们只关心矢量的方向而不是模。例如，在计算光照模型时，我们往往需要得到顶点的法线方向和光源方向，此时我们不关心这些矢量有多长。在这些情况下，我们就需要计算**单位矢量**（**unit vector**）。

单位矢量指的是那些模为1的矢量。单位矢量也被称为**被归一化的矢量**（**normalized vector**）。对任何给定的非零矢量，把它转换成单位矢量的过程就被称为**归一化**（**normalization**）。

给定任意非零矢量 $\mathbf{v}$ ，我们可以计算和 $\mathbf{v}$ 方向相同的单位矢量。在本书中，我们通过在矢量的头上添加一个戴帽符号来表示单位矢量，例如 $\hat{\mathbf{v}}$ 。为了对矢量进行归一化，我们可以用矢量除以该矢量的模来得到。公式如下：

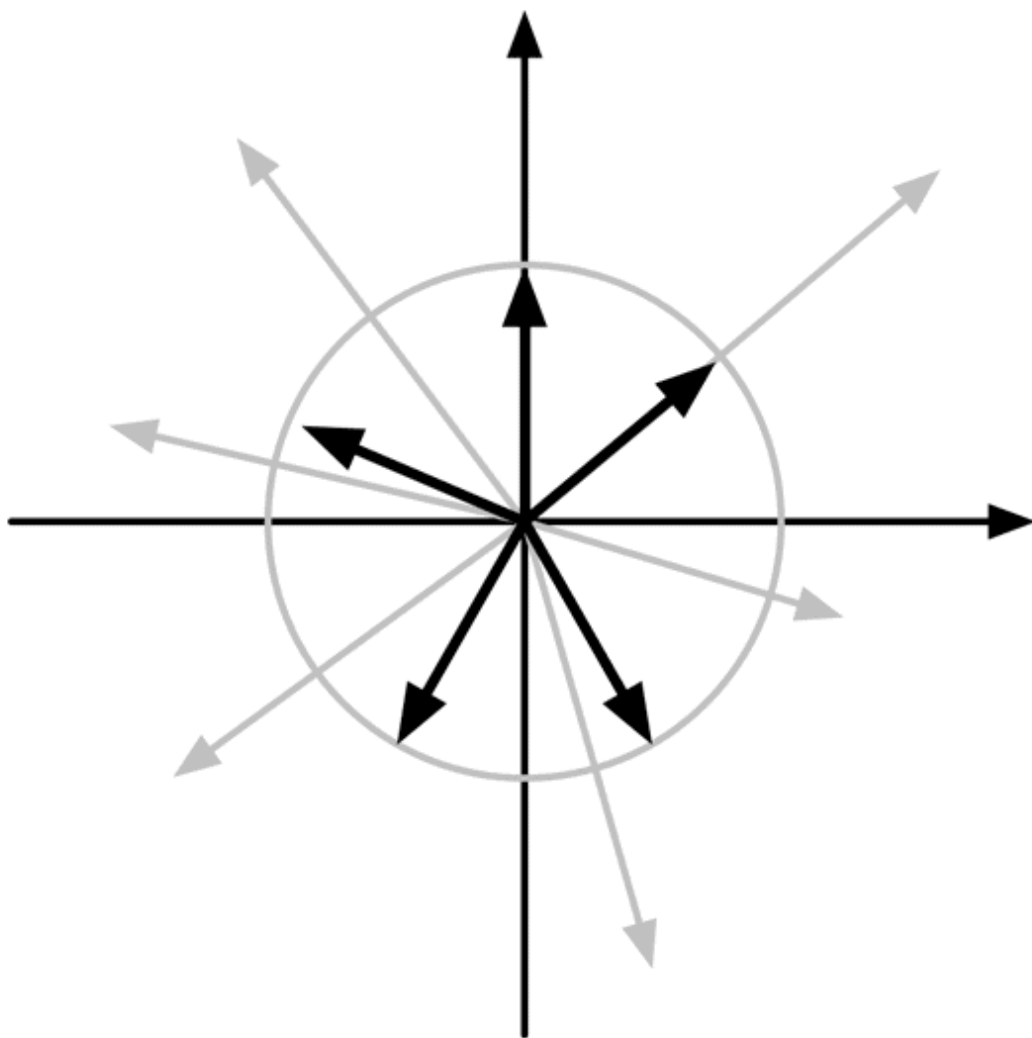
$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{|\mathbf{v}|}, \mathbf{v} \text{ 是任意非零矢量}$$

下面给出一些例子：

$$\frac{(3, -4)}{|(3, -4)|} = \frac{(3, -4)}{\sqrt{3^2 + (-4)^2}} = \frac{(3, -4)}{\sqrt{25}} = \frac{(3, -4)}{5} = \left(\frac{3}{5}, \frac{-4}{5}\right) = (0.6, -0.8)$$

**零矢量**（即矢量的每个分量值都为0，如 $\mathbf{v}=(0,0,0)$ ）是不可以被归一化的。这是因为做除法运算时分母不能为0。

从几何意义上看，对二维空间来说，我们可以画一个单位圆，那么单位矢量就可以是从圆心出发、到圆边界的矢量。在三维空间中，单位矢量就是从单位球的球心出发、到达球面的矢量。图4.21给出了二维空间内的一些单位矢量。



▲图4.21 二维空间的单位矢量都会落在单位圆上

需要注意的是，在后面的章节中我们将会不断遇到法线方向（也被称为法矢量）、光源方向等，这些矢量不一定是归一化后的矢量。由于我们的计算往往要求矢量是单位矢量，因此在使用前应先对这些矢量进行归一化运算。

## 5. 矢量的点积

矢量之间也可以进行乘法，但是和标量之间的乘法有很大不同。矢量的乘法有两种最常用的种类：**点积（dot product，也被称为内积，inner product）**和**叉积（cross product，也被称为外积，outer product）**。在本节中，我们将讨论第一种类型：点积。

读者可能认为上面几节的内容都很简单，“这些都显而易见嘛”。那么从这一节开始，我们就会遇到一些真正需要花费力气（真的只要一点点）去记忆的公式。幸运的是，绝大多数公式是有几何意义的，也就是说，我们可以通过画图的方式来理解和帮助记忆。

比仅仅记住这些公式更加重要的是，我们要真正理解它们是什么的。只有这样，我们才能在需要时想起来，“噢，这个需求我可以用这个公式来实现！”在我们编写Shader的过程中，通常程序接口都会提供这些公式的实现，因此我们往往不需要手工输入这些公式。例如，在Unity Shader中，我们可以直接使用形如dot(a, b)的代码来对两个矢量值进行点积的运算。

点积的名称来源于这个运算的符号： $\mathbf{a} \cdot \mathbf{b}$ 。中间的这个圆点符号是不可以省略的。点积的公式有两种形式，我们先来看第一种。两个三维矢量的点积是把两个矢量对应分量相乘后再取和，最后的结果是一个标量。

公式一：

$$\mathbf{a} \cdot \mathbf{b} = (a_x, a_y, a_z) \cdot (b_x, b_y, b_z) = a_x b_x + a_y b_y + a_z b_z$$

下面是一些例子：



$$(1,2,3) \cdot (0.5,4,2.5)=0.5+8+7.5=16$$

$$(-3,4,0) \cdot (5,-1,7)= -15+-4+0=-19$$

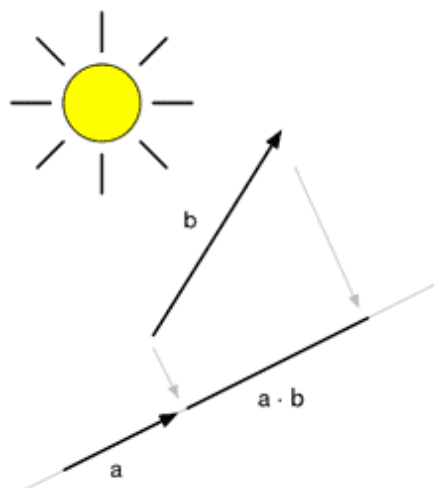
矢量的点积满足交换律，即 $\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a}$

点积的几何意义很重要，因为点积几乎应用到了图形学的各个方面。其中一个几何意义就是**投影（projection）**。

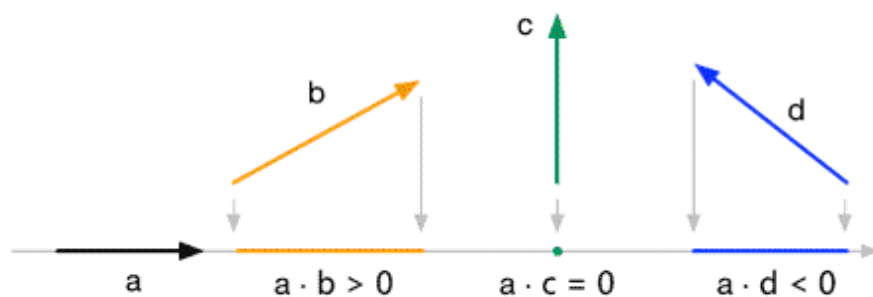
假设，有一个单位矢量 $\mathbf{a}$ 和另一个长度不限的矢量 $\mathbf{b}$ 。现在，我们希望得到 $\mathbf{b}$ 在平行于 $\mathbf{a}$ 的一条直线上的投影。那么，我们就可以使用点积 $\mathbf{a} \cdot \mathbf{b}$ 来得到 $\mathbf{b}$ 在 $\mathbf{a}$ 方向上的有符号的投影。

那么，投影到底是什么意思呢？这里给出一个通俗的解释。我们可以认为，现在有一个光源，它发出的光线是垂直于 $\mathbf{a}$ 方向的，那么 $\mathbf{b}$ 在 $\mathbf{a}$ 方向上的投影就是 $\mathbf{b}$ 在 $\mathbf{a}$ 方向上的影子，如图4.22所示。

需要注意的是，投影的值可能是负数。投影结果的正负号与 $\mathbf{a}$ 和 $\mathbf{b}$ 的方向有关：当它们的方向相反（夹角大于 $90^\circ$ ）时，结果小于0；当它们的方向互相垂直（夹角为 $90^\circ$ ）时，结果等于0；当它们的方向相同（夹角小于 $90^\circ$ ）时，结果大于0。图4.23给出了这3种情况的图示。



▲图4.22 矢量**b**在单位矢量**a**方向上的投影



▲图4.23 点积的符号

也就是说，点积的符号可以让我们知道两个矢量的方向关系。

那么，如果**a**不是一个单位矢量会如何呢？这很容易想到，任何两个矢量的点积 **$\mathbf{a} \cdot \mathbf{b}$** 等同于**b**在**a**方向上的投影值，再乘以**a**的长度。

点积具有一些很重要的性质，在Shader的计算中，我们会经常利用这些性质来帮助计算。

性质一：点积可结合标量乘法。

上面的“结合”是说，点积的操作数之一可以是另一个运算的结果，即矢量和标量相乘的结果。公式如下：

$$(ka) \cdot b = a \cdot (kb) = k(a \cdot b)$$

也就是说，对点积中其中一个矢量进行缩放的结果，相当于对最后的点积结果进行缩放。

性质二：点积可结合矢量加法和减法，和性质一类似。

这里的“结合”指的是，点积的操作数可以是矢量相加或相减后的结果。用公式表达就是：

$$a \cdot (b + c) = a \cdot b + a \cdot c$$

把上面的 $c$ 换成 $-c$ 就可以得到减法的版本。

性质三：一个矢量和本身进行点积的结果，是该矢量的模的平方。

这点可以很容易从公式验证得到：

$$\mathbf{v} \cdot \mathbf{v} = v_x v_x + v_y v_y + v_z v_z = |\mathbf{v}|^2$$

这意味着，我们可以直接利用点积来求矢量的模，而不需要使用模的计算公式。当然，我们需要对点积结果进行开平方的操作来得到真正的模。但很多情况下，我们只是想要比较两个矢量的长度大小，因此可以直接使用点积的结果。毕竟，开平方的运算需要消耗一定性能。

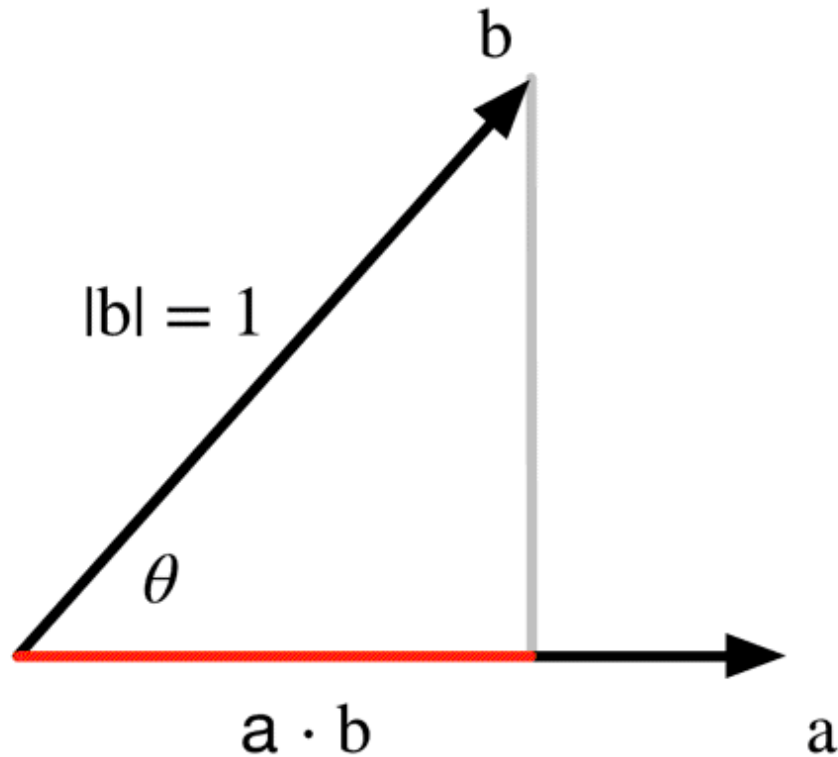
现在是时候来看点积的另一种表示方法了。这种方法是从三角代数的角度出发的，这种表示方法更加具有几何意义，因为它可以明确地强调出两个矢量之间的角度。

我们先直接给出第二个公式。

公式二：

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \cos \theta$$

初看之下，似乎和公式一没有什么联系，怎么会相等呢？我们先来看最简单的情况。假设，我们对两个单位矢量进行点积，即 $\hat{\mathbf{a}} \cdot \hat{\mathbf{b}}$ ，如图4.24所示。



▲图4.24 两个单位矢量进行点积

到了产生魔法的时间了！我们知道 $\hat{\mathbf{b}}$ 的模为1，且读者应该记得 $\cos\theta = \frac{\text{斜边}}{\text{斜边}}$ 。我们可以发现，图中 $\hat{\mathbf{a}} \cdot \hat{\mathbf{b}}$ 的结果刚好就是 $\cos\theta$ 对应的直角边。因此，由图4.24可以得到：

$$\hat{\mathbf{a}} \cdot \hat{\mathbf{b}} = \frac{\text{斜边}}{\text{斜边}} = \cos\theta$$

.这也就是说，两个单位矢量的点积等于它们之间夹角的余弦值。再应用性质一就可以得到公式二了：

$$\mathbf{a} \cdot \mathbf{b} = (|\mathbf{a}|\hat{\mathbf{a}}) \cdot (|\mathbf{b}|\hat{\mathbf{b}}) = |\mathbf{a}||\mathbf{b}|(\hat{\mathbf{a}} \cdot \hat{\mathbf{b}}) = |\mathbf{a}||\mathbf{b}|\cos\theta$$

也就是说，两个矢量的点积可以表示为两个矢量的模相乘，再乘以它们之间夹角的余弦值。从这个公式也可以看出，为什么计算投影时两个矢量的方向不同会得到不同符号的投影值：当夹角小于 $90^\circ$ 时， $\cos\theta>0$ ；当夹角等于 $90^\circ$ 时， $\cos\theta=0$ ；当夹角大于 $90^\circ$ 时， $\cos\theta<0$ 。

利用这个公式我们还可以求得两个向量之间的夹角（在 $0^\circ\sim 180^\circ$ ）：

$$\theta=\arccos(\hat{\mathbf{a}}\cdot\hat{\mathbf{b}}), \text{ 假设}\hat{\mathbf{a}}\text{和}\hat{\mathbf{b}}\text{是单位矢量。}$$

其中， $\arccos$ 是反余弦操作。

## 6. 矢量的叉积

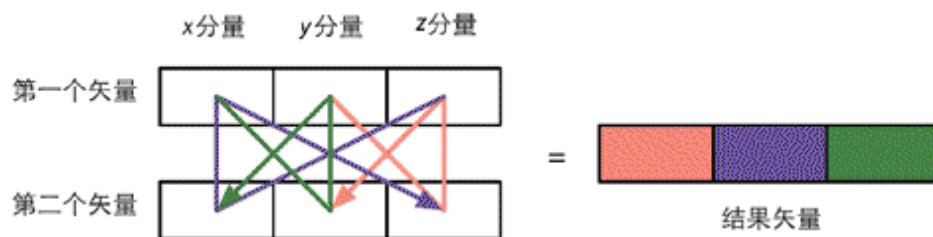
另一个重要的矢量运算就是**叉积**（**cross product**），也被称为**外积**（**outer product**）。与点积不同的是，矢量叉积的结果仍是一个矢量，而非标量。

和点积类似，叉积的名称来源于它的符号： $\mathbf{a}\times\mathbf{b}$ 。同样，这个叉号也是不可省略的。两个矢量的叉积可以用如下公式计算：

$$\mathbf{a}\times\mathbf{b}=(a_x, a_y, a_z)\times(b_x, b_y, b_z)=(a_yb_z-a_zb_y, a_zb_x-a_xb_z, a_xb_y-a_yb_x)$$

上面的公式看起来很复杂，但其实是有一定规律的。图4.25给出了这样的规律图示。





▲图4.25 三维矢量叉积的计算规律。不同颜色的线表示了计算结果矢量中对应颜色的分量的计算路径。以红色为例，即结果矢量的第一个分量，它是从第一个矢量的y分量出发乘以第二个矢量的z分量，再减去第一个矢量的z分量和第二矢量的y分量的乘积

例如：

$$(1,2,3) \times (-2,-1,4) = ((2)(4) - (3)(-1), (3)(-2) - (1)(4), (1)(-1) - (2)(-2))$$

$$= (8 - (-3), (-6) - 4, (-1) - (-4)) = (11, -10, 3)$$

需要注意的是，叉积不满足交换律，即  $\mathbf{a} \times \mathbf{b} \neq \mathbf{b} \times \mathbf{a}$ 。实际上，叉积是满足反交换律的，即  $\mathbf{a} \times \mathbf{b} = -(\mathbf{b} \times \mathbf{a})$ 。而且叉积也不满足结合律，即  $(\mathbf{a} \times \mathbf{b}) \times \mathbf{c} \neq \mathbf{a} \times (\mathbf{b} \times \mathbf{c})$ 。

从叉积的几何意义出发，我们可以更加深入地理解它的用处。对两个矢量进行叉积的结果会得到一个同时垂直于这两个矢量的新矢量。我们已经知道，矢量是由一个模和方向来定义的，那么这个新的矢量的模和方向是什么呢？

我们先来看它的模。 $\mathbf{a} \times \mathbf{b}$ 的长度等于 $\mathbf{a}$ 和 $\mathbf{b}$ 的模的乘积再乘以它们之间夹角的正弦值。公式如下：

$$|\mathbf{a} \times \mathbf{b}| = |\mathbf{a}| |\mathbf{b}| \sin \theta$$

读者可能已经发现，上述公式和点积的计算公式很类似，不同的是，这里使用的是正弦值。如果读者对中学数学还有记忆的话，可能还会发现，这和平行四边形的面积计算公式是一样的。如果你忘记了，没关系，我们在这里回忆一下。

如图4.26所示，我们使用**a**和**b**构建一个平行四边形。

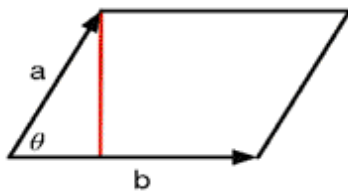
我们知道，平行四边形的面积可以使用 $|\mathbf{b}|h$ 来得到，即底乘以高。而 $h$ 又可以使用 $|\mathbf{a}|$ 和夹角 $\theta$ 来得到，即

$$A=|\mathbf{b}|h=|\mathbf{b}|(|\mathbf{a}|\sin\theta)=|\mathbf{a}||\mathbf{b}|\sin\theta=|\mathbf{a}\times\mathbf{b}|$$

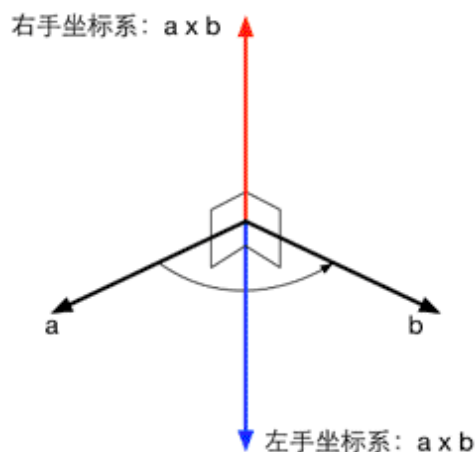
你可能会问，如果**a**和**b**平行（可以是方向完全相同，也可以是完全相反）怎么办，不就不能构建平行四边形了吗？我们可以认为构建出来的平行四边形面积为0，那么 $\mathbf{a}\times\mathbf{b}=\mathbf{0}$ 。注意，这里得到的是零向量，而不是标量0。

下面，我们来看结果矢量的方向。你可能会说：“方向？不是已经说了方向了嘛，就是和两个矢量都垂直就可以了啊。”但是，如果你仔细想一下就会发现，实际上我们有两个方向可以选择，这两个方向都和这两个矢量垂直。那么，我们要选择哪个方向呢？

这里就要和之前提到的左手坐标系和右手坐标系联系起来了，如图4.27所示。

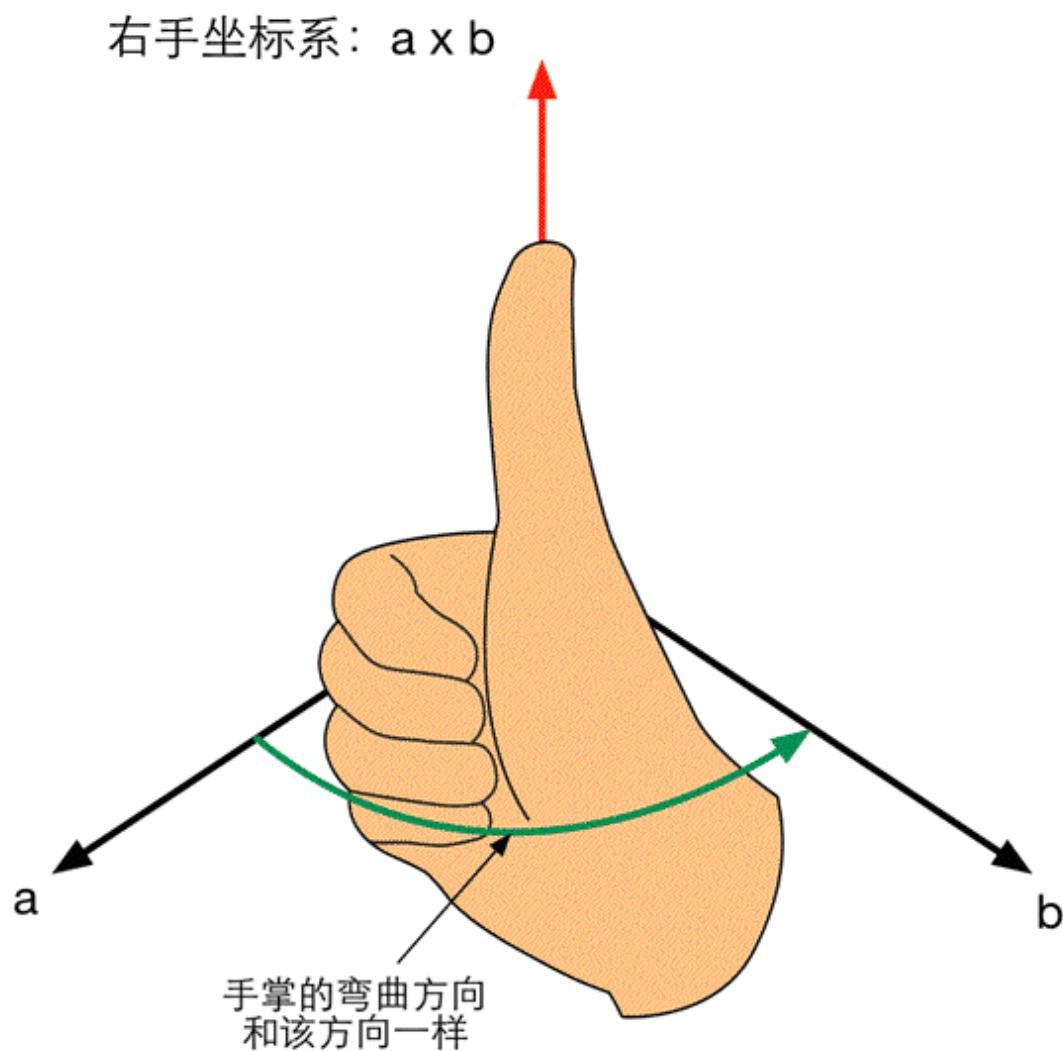


▲图4.26 使用矢量 $\mathbf{a}$ 和矢量 $\mathbf{b}$ 构建一个平行四边形



▲图4.27 分别使用左手坐标系和右手坐标系得到的叉积结果

这个结果是怎么得到的呢？来，举起你的双手！哦，不……先举起你的右手。在右手坐标系中， $\mathbf{a} \times \mathbf{b}$ 的方向将使用右手法则来判断。我们先想象把手心放在了 $\mathbf{a}$ 和 $\mathbf{b}$ 的尾部交点处，然后张开你的手掌让手掌方向和 $\mathbf{a}$ 的方向重合，再弯曲你的四指让它们向 $\mathbf{b}$ 的方向靠拢，最后伸出你的大拇指！大拇指指向的方向就是右手坐标系中 $\mathbf{a} \times \mathbf{b}$ 的方向了。如果你实在不明白怎么摆放和扭动你的手，那么就看图4.28好了。



▲图4.28 使用右手法则判断右手坐标系中 $\mathbf{a} \times \mathbf{b}$ 的方向

同理，我们可以使用左手法则来判断左手坐标系中 $\mathbf{a} \times \mathbf{b}$ 的方向。赶紧举起你的左手试试吧（你可能会发现这个姿势比较扭曲）！

需要注意的是，虽然看起来左右手坐标系的选择会影响叉积的结果，但这仅仅是“看起来”而已。从叉积的数学表达式可以发现，使用

左手坐标系还是右手坐标系不会对计算结果产生任何影响，它影响的只是数字在三维空间中的视觉化表现而已。当从右手坐标系转换为左手坐标系时，所有点和矢量的表达和计算方式都会保持不变，只是当呈现到屏幕上时，我们可能会发现，“咦，怎么图像反过来了！”。当我们想要两个坐标系达到同样的视觉效果时，可能就需要改变一些数学运算公式，这不在本书的范畴内。有兴趣的读者可以参考本章的扩展阅读部分。

那么，叉积到底有什么用呢？最常见的一个应用就是计算垂直于一个平面、三角形的矢量。另外，还可以用于判断三角面片的朝向。读者可以在本节的练习题中找到这些应用。

### 4.3.3 练习题

又到了做练习的时候了，大家是不是都很激动！那么，赶紧拿起笔、拿起纸开始吧！

#### 1. 是非题

(1) 一个矢量的大小不重要，我们只需要在正确的位置把它画出来就可以了。

(2) 点可以认为是位置矢量，这是通过把矢量的尾固定在原点得到的。

(3) 选择左手坐标系还是右手坐标系很重要，因为这会影响叉积的计算。

#### 2. 计算下面的矢量运算：

(1)  $|(2,7,3)|$

(2)  $2.5(5,4,10)$

(3)  $\frac{(3,4)}{2}$

(4) 对 $(5,12)$ 进行归一化

(5)  $(1,1,1)$ 进行归一化

(6)  $(7,4)+(3,5)$

(7)  $(9,4,13)-(15,3,11)$

3. 假设，场景中有一个光源，位置在 $(10,13,11)$ 处，还有一个点 $(2,1,1)$ ，那么光源距离该点的距离是多少？

4. 计算下面的矢量运算：

(1)  $(4,7) \cdot (3,9)$

(2)  $(2,5,6) \cdot (3,1,2) - 10$

(3)  $0.5(-3,4) \cdot (-2,5)$

(4)  $(3,-1,2) \times (-5,4,1)$

(5)  $(-5,4,1) \times (3,-1,2)$

5. 已知矢量 $\mathbf{a}$ 和矢量 $\mathbf{b}$ ， $\mathbf{a}$ 的模为4， $\mathbf{b}$ 的模为6，它们之间的夹角为 $60^\circ$ 。计算：



(1)  $\mathbf{a} \cdot \mathbf{b}$

(2)  $|\mathbf{a} \times \mathbf{b}|$ 提示:  $\sin 60^\circ = \frac{\sqrt{3}}{2} \approx 0.866$ ,  $\cos 60^\circ = \frac{1}{2} = 0.5$ 。

6. 假设, 场景中有一个NPC, 它位于点 $p$ 处, 它的前方(forward)可以使用矢量 $\mathbf{v}$ 来表示。

(1) 如果现在玩家运动到了点 $x$ 处, 那么如何判断玩家是在NPC的前方还是后方? 请使用数学公式来描述你的答案。提示: 使用点积。

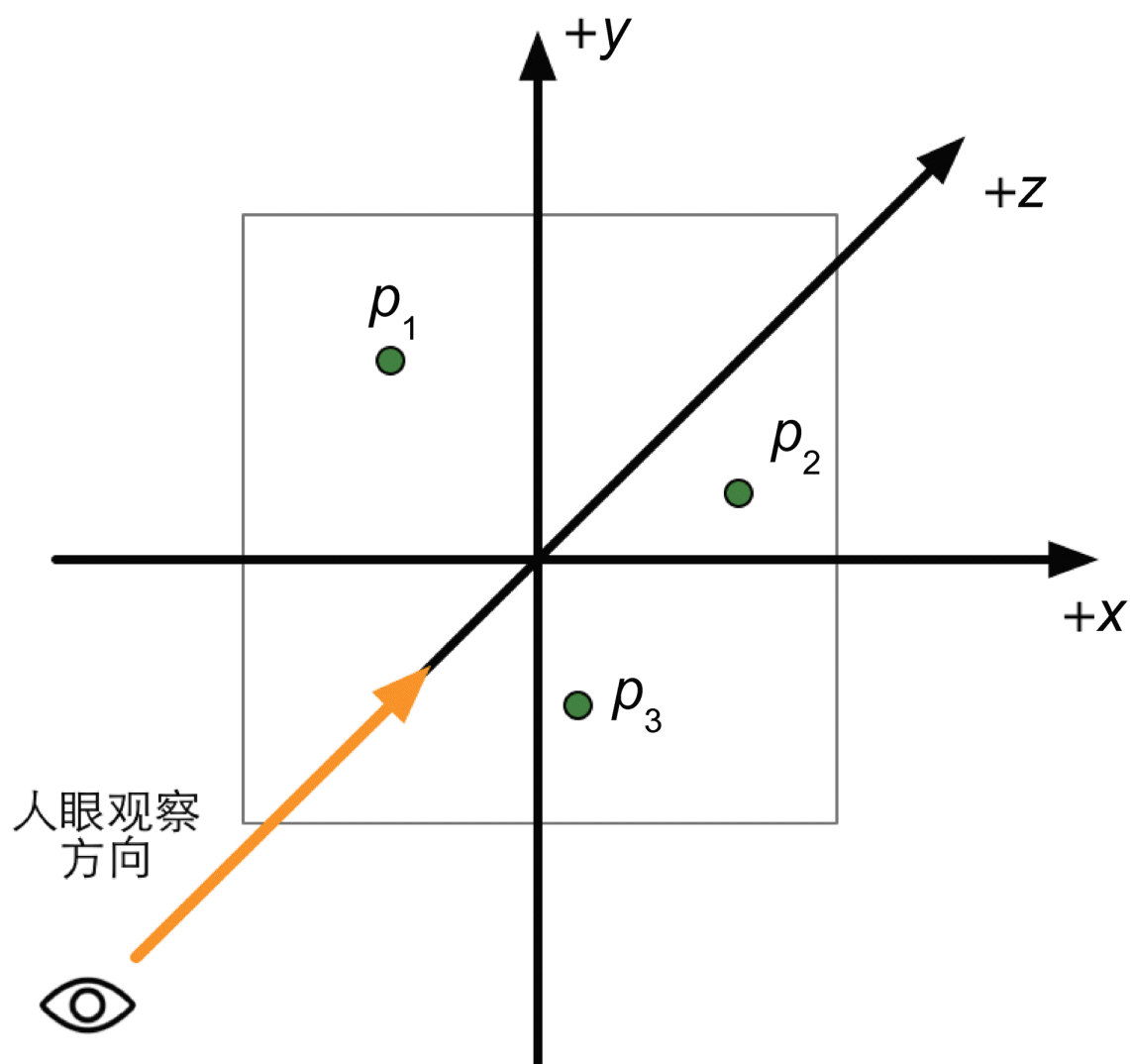
(2) 使用你在a中提到的方法, 代入 $p=(4,2), \mathbf{v}=(-3,4), x=(10,6)$ 来验证你的答案。

(3) 现在, 游戏有了新的需求: NPC只能观察到有限的视角范围, 这个视角的角度是 $\phi$ , 也就是说NPC最多只能看到它前方左侧或右侧 $\frac{\phi}{2}$ 角度内的物体。那么, 我们如何通过点积来判断NPC是否可以看点 $x$ 呢?

(4) 在c的条件基础上, 策划又有了新的需求: NPC的观察距离也有了限制, 它只能看到固定距离内的对象, 现在又如何判断呢?

7. 在渲染中我们时常会需要判断一个三角面片是正面还是背面, 这可以通过判断三角形的3个顶点在当前空间中是顺时针还是逆时针排列来得到。给定三角形的3个顶点 $p_1$ 、 $p_2$ 和 $p_3$ , 如何利用叉积来判断这3个点的顺序是顺时针还是逆时针? 假设我们使用的是左手坐标系, 且

$p_1$ 、 $p_2$ 和 $p_3$ 都位于 $xy$ 平面（即它们的 $z$ 分量均为0），人眼位于 $z$ 轴的负方向上，向 $z$ 轴正方向观察，如图4.29所示。



▲图4.29 三角形的三个顶点位于 $xy$ 平面上，人眼位于 $z$ 轴负方向，向 $z$ 轴正方向观察

## 4.4 矩阵

不幸的是，没有人能告诉你母体（**matrix**）究竟是什么。你需要自己去发现它。

——电影《黑客帝国》（英文名：The Matrix）

**矩阵**，英文名是**matrix**。如果你用翻译软件去查**matrix**这个单词的翻译，就会发现它还有一个意思就是母体。事实上，很多人都不知道，那部具有跨时代意义的电影《黑客帝国》的英文名就是《The Matrix》。在电影《黑客帝国》中，母体是一个庞大的虚拟系统，它看似虚无缥缈，但又连接万物。这一点和矩阵有异曲同工之妙。

没有人敢否认矩阵在三维数学中的重要性，事实上矩阵在整个线性代数的世界中都扮演了举足轻重的角色。在三维数学中，我们通常会使用矩阵来进行变换。一个矩阵可以把一个矢量从一个坐标空间转换到另一个坐标空间。在第2章渲染流水线中，我们就看到了很多坐标变换，例如在顶点着色器中我们需要把顶点坐标从模型空间变换到齐次裁剪坐标系中。而在这一章中，我们先来认识一下矩阵这个概念。

那么，现在我们就来看一下，这些放在一个小括号里的数字怎么就这么重要呢？为什么数学家们都喜欢用这个小东西来搞出这么多名堂呢？

#### 4.4.1 矩阵的定义

相信很多读者都见过矩阵的真容，例如像下面这个样子：

$$\begin{bmatrix} 1 & 0.5 & 3 & 2 \\ 2.3 & 5 & \sqrt{3} & 10 \\ 4 & 8 & 11 & 5 \end{bmatrix}$$

从它的外观上来看，就是一个长方形的网格，每个格子里放了一个数字。的确，矩阵就是这么简单：它是由 $m \times n$ 个标量组成的长方形数组。在上面的式子中，我们是用方括号来围住矩阵中的数字，而一些其他的资料可能会使用圆括号或花括号来表示，这都是等价的。

既然是网格结构，就意味着矩阵有**行**（row）**列**（column）之分。例如上面的例子就是一个 $3 \times 4$ 的矩阵，它有三行四列。据此，我们可以给出矩阵的一般表达式。以 $3 \times 3$ 的矩阵为例，它可以写成：

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}$$

$m_{ij}$ 表明了这个元素在矩阵**M**的第*i*行、第*j*列。

这样看起来矩阵也没什么神秘的嘛。但是，越简单的东西往往越厉害，这也是数学的魅力所在。

#### 4.4.2 和矢量联系起来

前面说到，矢量其实就是一个数组，而矩阵也是一个数组。既然都是数组，那就是一家人了！我们很容易想到，我们可以用矩阵来表示矢量。实际上，矢量可以看成是 $n \times 1$ 的**列矩阵**（**column matrix**）或 $1 \times n$ 的**行矩阵**（**row matrix**），其中*n*对应了矢量的维度。例如，矢量 $\mathbf{v}=(3,8,6)$ 可以写成行矩阵

$$[3 \quad 8 \quad 6]$$

或列矩阵

$$\begin{bmatrix} 3 \\ 8 \\ 6 \end{bmatrix}$$

为什么我们要把矢量和矩阵联系在一起呢？这是为了可以让矢量像一个矩阵一样一起参与矩阵运算。这在空间变换中将非常有用。

到现在，使用行矩阵还是列矩阵来表示矢量看起来是没什么分别的。的确，我们可以根据自己的喜好来选择表示方法，但是，如果要和矩阵一起参与乘法运算时，这种选择会影响我们的书写顺序和结果。这正是我们下面要讲到的。

### 4.4.3 矩阵运算

矩阵这个家伙看起来比矢量要庞大很多，那么它的运算是不是很复杂呢？答案是肯定的。但是，幸运的是在写Shader的过程中，我们只需要和很简单的一部分运算打交道。

#### 1. 矩阵和标量的乘法

和矢量类似，矩阵也可以和标量相乘，它的结果仍然是一个相同维度的矩阵。它们之间的乘法非常简单，就是矩阵的每个元素和该标量相乘。以3×3的矩阵为例，其公式如下：

$$k\mathbf{M} = \mathbf{M}k = k \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} = \begin{bmatrix} km_{11} & km_{12} & km_{13} \\ km_{21} & km_{22} & km_{23} \\ km_{31} & km_{32} & km_{33} \end{bmatrix}$$

#### 2. 矩阵和矩阵的乘法

两个矩阵的乘法也很简单，它们的结果会是一个新的矩阵，并且这个矩阵的维度和两个原矩阵的维度都有关系。

一个 $r \times n$ 的矩阵 $\mathbf{A}$ 和一个 $n \times c$ 的矩阵 $\mathbf{B}$ 相乘，它们的结果 $\mathbf{AB}$ 将会是一个 $r \times c$ 大小的矩阵。请读者注意它们的行列关系，第一个矩阵的列数必须和第二个矩阵的行数相同，它们相乘得到的矩阵的行数是第一个矩阵的行数，而列数是第二个矩阵的列数。例如，如果矩阵 $\mathbf{A}$ 的维度是 $4 \times 3$ ，矩阵 $\mathbf{B}$ 的维度是 $3 \times 6$ ，那么 $\mathbf{AB}$ 的维度就是 $4 \times 6$ 。

如果两个矩阵的行列不满足上面的规定怎么办？那么很抱歉，这两个矩阵就不能相乘，因为它们之间的乘法是没有被定义的（当然，读者完全可以自己定义一种新的乘法，但是数学家们会不会买账就不一定了）。那么为什么会有上面的规定呢？等我们理解了矩阵乘法的操作过程自然就会明白。

我们先给出看起来很复杂难懂（当给出直观的表式后读者会发现其实它没那么难懂）的数学表达式：设有 $r \times n$ 的矩阵 $\mathbf{A}$ 和一个 $n \times c$ 的矩阵 $\mathbf{B}$ ，它们相乘会得到一个 $r \times c$ 的矩阵 $\mathbf{C} = \mathbf{AB}$ 。那么， $\mathbf{C}$ 中的每一个元素 $c_{ij}$ 等于 $\mathbf{A}$ 的第 $i$ 行所对应的矢量和 $\mathbf{B}$ 的第 $j$ 列所对应的矢量进行矢量点乘的结果，即

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^n a_{ik}b_{kj}$$

看起来很复杂对吗？但是，我们可以用一个更简单的方式来解释：对于每个元素 $c_{ij}$ ，我们找到 $\mathbf{A}$ 中的第 $i$ 行和 $\mathbf{B}$ 中的第 $j$ 列，然后把它们的对应元素相乘后再加起来，这个和就是 $c_{ij}$ 。

一种更直观的方式如图4.30所示。假设**A**的大小是4×2，**B**的大小是2×4，那么如果要计算**C**的元素 $c_{23}$ 的话，先找到对应的行矩阵和列矩阵，即**A**中的第2行和**B**中的第3列，把它们进行矢量点积后就可以得到结果值。因此， $c_{23}=a_{21}b_{13}+a_{22}b_{23}$ 。

$$\begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix}
 \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

▲ 图4.30 计算 $c_{23}$ 的过程

在Shader的计算中，我们更多的是使用4×4矩阵来运算的。

矩阵乘法满足一些性质。

性质一：矩阵乘法并不满足交换律。

也就是说，通常情况下：



$$\mathbf{AB} \neq \mathbf{BA}$$

性质二：矩阵乘法满足结合律。

也就是说，

$$(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$$

矩阵乘法的结合律可以扩展到更多矩阵的相乘。例如，

$$\mathbf{ABCDE} = ((\mathbf{A}(\mathbf{BC}))\mathbf{D})\mathbf{E} = (\mathbf{AB})(\mathbf{CD})\mathbf{E}$$

读者可根据矩阵乘法的定义很轻松地验证上述结论。

#### 4.4.4 特殊的矩阵

有一些特殊的矩阵类型在Shader中经常见到。这些特殊的矩阵往往具有一些重要的性质。

##### 1. 方块矩阵

**方块矩阵**（**square matrix**），简称方阵，是指那些行和列数目相等的矩阵。在三维渲染里，最常使用的就是 $3 \times 3$ 和 $4 \times 4$ 的方阵。

方阵之所以值得单独拿出来讲，是因为矩阵的一些运算和性质是只有方阵才具有的。例如，**对角元素**（**diagonal elements**）。方阵的对角元素指的是行号和列号相等的元素，例如 $m_{11}$ 、 $m_{22}$ 、 $m_{33}$ 等。如果把方阵看成一个正方形的话，这些元素排列在正方形的对角线上，这也是它们名字的由来。如果一个矩阵除了对角元素外的所有元素都为0，

那么这个矩阵就叫做**对角矩阵**（**diagonal matrix**）。例如，下面就是一个4×4的对角矩阵：

$$\begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 7 \end{bmatrix}$$

## 2. 单位矩阵

一个特殊的对角矩阵是**单位矩阵**（**identity matrix**），用 $I_n$ 来表示。一个3×3的单位矩阵如下：

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

为什么要为这种矩阵单独起一个名字呢？这是因为，任何矩阵和它相乘的结果都还是原来的矩阵。也就是说，

$$\mathbf{MI} = \mathbf{IM} = \mathbf{M}$$

这就跟标量中的数字1一样！

## 3. 转置矩阵

**转置矩阵**（**transposed matrix**）实际是对原矩阵的一种运算，即转置运算。给定一个 $r \times c$ 的矩阵 $\mathbf{M}$ ，它的转置可以表示成 $\mathbf{M}^T$ ，这是一个 $c \times r$ 的矩阵。转置矩阵的计算非常简单，我们只需要把原矩阵翻转一下即可。也就是说，原矩阵的第 $i$ 行变成了第 $i$ 列，而第 $j$ 列变成了第 $j$ 行。数学公式是：

$$\mathbf{M}_{ij}^T = \mathbf{M}_{ji}$$

例如,

$$\begin{bmatrix} 6 & 2 & 10 & 3 \\ 7 & 5 & 4 & 9 \end{bmatrix}^T = \begin{bmatrix} 6 & 7 \\ 2 & 5 \\ 10 & 4 \\ 3 & 9 \end{bmatrix}$$

对于行矩阵和列矩阵来说, 我们可以使用转置操作来转换行列矩阵:

$$[x \ y \ z]^T = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}^T = [x \ y \ z]$$

转置矩阵也有一些常用的性质。

性质一: 矩阵转置的转置等于原矩阵。

很容易理解, 我们把一个矩阵翻转一下后再翻转一下, 等于没有对矩阵做任何操作。即

$$(\mathbf{M}^T)^T = \mathbf{M}$$

性质二: 矩阵串接的转置, 等于反向串接各个矩阵的转置。

用公式表示就是:

$$(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T$$

该性质同样可以扩展到更多矩阵相乘的情况。

#### 4. 逆矩阵

**逆矩阵 (inverse matrix)** 大概是本书讲到的关于矩阵最复杂的一种操作了。不是所有的矩阵都有逆矩阵，第一个前提就是，该矩阵必须是一个方阵。

给定一个方阵 $\mathbf{M}$ ，它的逆矩阵用 $\mathbf{M}^{-1}$ 来表示。逆矩阵最重要的性质就是，如果我们把 $\mathbf{M}$ 和 $\mathbf{M}^{-1}$ 相乘，那么它们的结果将会是一个单位矩阵。也就是说，

$$\mathbf{M}\mathbf{M}^{-1} = \mathbf{M}^{-1}\mathbf{M} = \mathbf{I}$$

前面说了，并非所有的方阵都有对应的逆矩阵。一个明显的例子就是一个所有元素都为0的矩阵，很显然，任何矩阵和它相乘都会得到一个零矩阵，即所有的元素仍然都是0。如果一个矩阵有对应的逆矩阵，我们就说这个矩阵是**可逆的 (invertible)** 或者说是**非奇异的**

**(nonsingular)**；相反的，如果一个矩阵没有对应的逆矩阵，我们就说它是**不可逆的 (noninvertible)** 或者说是**奇异的 (singular)**。

那么如何判断一个矩阵是否是可逆的呢？简单来说，如果一个矩阵的**行列式 (determinant)** 不为0，那么它就是可逆的。关于矩阵的行列式是什么以及如何求解一个矩阵的逆矩阵，可以参见本章的扩展阅读部分。由于这部分内容涉及较多计算和其他定义，本书不再赘述。在写Shader的过程中，这些矩阵通常可以通过调用第三方库（如C++数

学库Eigen) 来直接求得，不需要开发者手动计算。在Unity中，重要变换矩阵的逆矩阵Unity也提供了相应的变量供我们使用。关于这些Unity内置的矩阵，读者可以在本章的4.8节找到更详细的解释。

逆矩阵有很多非常重要的性质。

性质一：逆矩阵的逆矩阵是原矩阵本身。

假设矩阵 $\mathbf{M}$ 是可逆的，那么

$$(\mathbf{M}^{-1})^{-1}=\mathbf{M}$$

性质二：单位矩阵的逆矩阵是它本身。

即

$$\mathbf{I}^{-1}=\mathbf{I}$$

性质三：转置矩阵的逆矩阵是逆矩阵的转置。

即

$$(\mathbf{M}^T)^{-1}=(\mathbf{M}^{-1})^T$$

性质四：矩阵串接相乘后的逆矩阵等于反向串接各个矩阵的逆矩阵。

即

$$(\mathbf{AB})^{-1}=\mathbf{B}^{-1}\mathbf{A}^{-1}$$

这个性质也可以扩展到更多矩阵的连乘，如：

$$(\mathbf{ABCD})^{-1}=\mathbf{D}^{-1}\mathbf{C}^{-1}\mathbf{B}^{-1}\mathbf{A}^{-1}$$

逆矩阵是具有几何意义的。我们知道一个矩阵可以表示一个变换（详见4.5节），而逆矩阵允许我们还原这个变换，或者说是计算这个变换的反向变换。因此，如果我们使用变换矩阵 $\mathbf{M}$ 对矢量 $\mathbf{v}$ 进行了一次变换，然后再使用它的逆矩阵 $\mathbf{M}^{-1}$ 进行另一次变换，那么我们会得到原来的矢量。这个性质可以使用矩阵乘法的结合律很容易地进行证明：

$$\mathbf{M}^{-1}(\mathbf{M}\mathbf{v})=(\mathbf{M}^{-1}\mathbf{M})\mathbf{v}=\mathbf{I}\mathbf{v}=\mathbf{v}$$

## 5. 正交矩阵

另一个特殊的方阵是**正交矩阵**（**orthogonal matrix**）。正交是矩阵的一种属性。如果一个方阵 $\mathbf{M}$ 和它的转置矩阵的乘积是单位矩阵的话，我们就说这个矩阵是**正交的**（**orthogonal**）。反过来也是成立的。也就是说，矩阵 $\mathbf{M}$ 是正交的等价于：

$$\mathbf{M}\mathbf{M}^T=\mathbf{M}^T\mathbf{M}=\mathbf{I}$$

读者可能已经看出来，上式和我们在上一节讲到的逆矩阵时遇到的公式很像。把这两个公式结合起来，我们就可以得到一个重要的性质，即如果一个矩阵是正交的，那么它的转置矩阵和逆矩阵是一样的。也就是说，矩阵 $\mathbf{M}$ 是正交的等价于：

$$\mathbf{M}^T=\mathbf{M}^{-1}$$

这个式子非常有用，因为在三维变换中我们经常会需要使用逆矩阵来求解反向的变换。而逆矩阵的求解往往计算量很大，但转置矩阵却非常容易求解：我们只需要把矩阵翻转一下就可以了。那么，我们如何提前判断一个矩阵是否是正交矩阵呢？读者可能会说，判断  $\mathbf{M}\mathbf{M}^T=\mathbf{I}$  是否成立就可以了嘛！但是，求解这样一个表达式无疑是需要一定计算量的，这些计算量可能和直接求解逆矩阵无异。而且，如果我们判断出来这不是一个正交矩阵，那么这些花在验证是否是正交矩阵上的计算就浪费了。因此，我们更想不需要计算，而仅仅根据一个矩阵的构造过程来判断这个矩阵是否是正交矩阵。为此，我们需要来了解正交矩阵的几何意义。

我们来看一下对于3×3的正交矩阵有什么特点。根据正交矩阵的定义，我们有：

$$\begin{aligned} \mathbf{M}^T\mathbf{M} &= \begin{bmatrix} - & c_1 & - \\ - & c_2 & - \\ - & c_3 & - \end{bmatrix} \begin{bmatrix} | & | & | \\ c_1 & c_2 & c_3 \\ | & | & | \end{bmatrix} \\ &= \begin{bmatrix} c_1 \cdot c_1 & c_1 \cdot c_2 & c_1 \cdot c_3 \\ c_2 \cdot c_1 & c_2 \cdot c_2 & c_2 \cdot c_3 \\ c_3 \cdot c_1 & c_3 \cdot c_2 & c_3 \cdot c_3 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \mathbf{I} \end{aligned}$$

这样，我们就有了9个等式：

$$\mathbf{c}_1 \cdot \mathbf{c}_1 = 1, \quad \mathbf{c}_1 \cdot \mathbf{c}_2 = 0, \quad \mathbf{c}_1 \cdot \mathbf{c}_3 = 0$$

$$\mathbf{c}_2 \cdot \mathbf{c}_1 = 0, \quad \mathbf{c}_2 \cdot \mathbf{c}_2 = 1, \quad \mathbf{c}_2 \cdot \mathbf{c}_3 = 0$$

$$\mathbf{c}_3 \cdot \mathbf{c}_1 = 0, \quad \mathbf{c}_3 \cdot \mathbf{c}_2 = 0, \quad \mathbf{c}_3 \cdot \mathbf{c}_3 = 1$$



我们可以得到以下结论：

- 矩阵的每一行，即 $\mathbf{c}_1$ 、 $\mathbf{c}_2$ 和 $\mathbf{c}_3$ 是单位矢量，因为只有这样它们与自己的点积才能是1；
- 矩阵的每一行，即 $\mathbf{c}_1$ 、 $\mathbf{c}_2$ 和 $\mathbf{c}_3$ 之间互相垂直，因为只有这样它们之间的点积才能是0。
- 上述两条结论对矩阵的每一列同样适用，因为如果 $\mathbf{M}$ 是正交矩阵的话， $\mathbf{M}^T$ 也会是正交矩阵。

也就是说，如果一个矩阵满足上面的条件，那么它就是一个正交矩阵。读者可以注意到，一组标准正交基（定义详见4.2.2节）可以精确地满足上述条件。在4.6.2节中，我们会使用坐标空间的基矢量来构建用于空间变换的矩阵。因此，如果这些基矢量是一组标准正交基的话（例如只存在旋转变换），那么我们就可以直接使用转置矩阵来求得该变换的逆变换。

读者：我被标准正交、正交这些概念搞混了，可以再说明一下是什么意思吗？

我们：读者应该已经知道，一个坐标空间需要指定一组基矢量，也就是我们理解的坐标轴。如果这些基矢量之间是互相垂直的，那么我们就把它们称为是一组**正交基**（**orthogonal basis**）。但是，它们的长度并不要求一定是1。如果它们的长度的确是1的话，我们就说它们是一组**标准正交基**（**orthonormal basis**）。因此，一个正交矩阵的行和列之间分别构成了一组标准正交基。但是，如果我们使用一组正交基来构建一个矩阵的话，这个矩阵可能就不是一个正交矩阵，因为这些基矢量的长度可能不为1，也就是说它们不是标准正交基。

#### 4.4.5 行矩阵还是列矩阵

我们已经了解了足够多的数学概念，但在学习矩阵的几何意义之前，我们有必要说明一下行矩阵和列矩阵的问题。

在前面的章节中我们讲到，可以把一个矢量转换成一个行矩阵或是列矩阵。它们本身是没有区别的，但是，当我们需要把它和另一个矩阵相乘时，就会出现一些差异。

假设有一个矢量 $\mathbf{v}=(x,y,z)$ ，我们可以把它转换成行矩阵 $\mathbf{v}=[xyz]$ 或列矩阵 $\mathbf{v}=[x\ y\ z]^T$ （这里使用了转置符号来避免列矩阵在我们的这一行中显得太高）。现在，有另一个矩阵 $\mathbf{M}$ ：

$$\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}$$

那么 $\mathbf{M}$ 分别和行矩阵以及列矩阵相乘后会是什么结果呢？我们先来看 $\mathbf{M}$ 和行矩阵的相乘。由矩阵乘法的定义可知，我们需要把行矩阵放在 $\mathbf{M}$ 的左边（还记得吗，矩阵乘法要求两个矩阵的行列数满足一定条件），即

$$\mathbf{vM} = [xm_{11} + ym_{21} + zm_{31} \quad xm_{12} + ym_{22} + zm_{32} \quad xm_{13} + ym_{23} + zm_{33}]$$

而如果要和列矩阵相乘的话，结果是：

$$\mathbf{Mv} = \begin{bmatrix} xm_{11} + ym_{12} + zm_{13} \\ xm_{21} + ym_{22} + zm_{23} \\ xm_{31} + ym_{32} + zm_{33} \end{bmatrix}$$

读者认真对比就会发现，结果矩阵除了行列矩阵的区别外，里面的元素也是不一样的。这就意味着，在和矩阵相乘时选择行矩阵还是列矩阵来表示矢量是非常重要的，因为这决定了矩阵乘法的书写次序和结果值。

在Unity中，常规做法是把矢量放在矩阵的右侧，即把矢量转换成列矩阵来进行运算。因此，在本书后面的内容中，如无特殊情况，我们都将使用列矩阵。这意味着，我们的矩阵乘法通常都是右乘，例如：

$$\mathbf{CBA}\mathbf{v} = (\mathbf{C}(\mathbf{B}(\mathbf{A}\mathbf{v})))$$

使用列向量的结果是，我们的阅读顺序是从右到左，即先对 $\mathbf{v}$ 使用 $\mathbf{A}$ 进行变换，再使用 $\mathbf{B}$ 进行变换，最后使用 $\mathbf{C}$ 进行变换。

上面的计算等价于下面的行矩阵运算：

$$\mathbf{v}\mathbf{A}^T\mathbf{B}^T\mathbf{C}^T = (((\mathbf{v}\mathbf{A}^T)\mathbf{B}^T)\mathbf{C}^T)$$

如果你还是不能明白上面的含义，可以参见练习题3。

#### 4.4.6 练习题

1. 判断下面矩阵的乘法是否存在。如果存在，计算它们的乘积。

$$(1) \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} -1 & 5 \\ 0 & 2 \end{bmatrix}$$

$$(2) \begin{bmatrix} 2 & 4 & 3 \\ 2 & 1 & 4 \end{bmatrix} \begin{bmatrix} -1 & 5 \\ 0 & 2 \\ 3 & 10 \\ 4 & 5 \end{bmatrix}$$

$$(3) \begin{bmatrix} 1 & -2 & 3 \\ 5 & 1 & 4 \\ 6 & 0 & 3 \end{bmatrix} \begin{bmatrix} -5 \\ 4 \\ 8 \end{bmatrix}$$

2. 判断下面的矩阵是否是正交矩阵。

$$(1) \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

$$(2) \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$(3) \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

3. 给定一个矢量(3,2,6)，分别把它当成行矩阵和列矩阵与下面的矩阵相乘。考虑两种情况下得到的矢量结果是否一样。如果不一样，考虑如何得到相同的结果。

$$(1) \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$(2) \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & -3 \\ 0 & 0 & 3 \end{bmatrix}$$

$$(3) \begin{bmatrix} 2 & -1 & 3 \\ -1 & 5 & -3 \\ 3 & -3 & 4 \end{bmatrix}$$

## 4.5 矩阵的几何意义：变换

关于矩阵，很多困扰初学者的问题都是类似的：

- 点和矢量都可以在图像中画出来，那么矩阵可以吗？
- 我听说矩阵和线性变换、仿射变换有关，这些变换到底是什么意思呢？
- 我总是听到齐次坐标这个名词，它是什么意思呢？
- 变换和矩阵的关系又是什么呢？或者说，给定一个变换，我如何得到它对应的矩阵呢？

在学习完本节后，希望读者们能够回答出这些问题。

对于第一个问题，在三维渲染中矩阵可以可视化吗？幸运的是，答案是肯定的，这个可视化的结果就是变换。因此，如果读者在后面的内容中看到了一个矩阵，那么你可以认为自己看到的就是一个变换（当然，在线性代数中矩阵的用处不仅是用于变换，但本书的讨论范围仅在于此）。

在游戏的世界中，这些变换一般包含了旋转、缩放和平移。游戏开发人员希望给定一个点或矢量，再给定一个变换（例如把点平移到

另一个位置，把矢量的方向旋转30°等），就可以通过某个数学运算来求得新的点和矢量。聪明的先人们发现，可以使用矩阵来完美地解决这个问题。那么问题就变成了，我们如何使用矩阵来表示这些变换？

### 4.5.1 什么是变换

**变换（transform）**，指的是我们把一些数据，如点、方向矢量甚至是颜色等，通过某种方式进行转换的过程。在计算机图形学领域，变换非常重要。尽管通过变换我们能够进行的操作是有限的，但这些操作已经足够奠定变换在图形学领域举足轻重的地位了。

我们先来看一个非常常见的变换类型——**线性变换（linear transform）**。线性变换指的是那些可以保留矢量加和标量乘的变换。用数学公式来表示这两个条件就是：

$$\mathbf{f}(\mathbf{x}) + \mathbf{f}(\mathbf{y}) = \mathbf{f}(\mathbf{x} + \mathbf{y})$$

$$k\mathbf{f}(\mathbf{x}) = \mathbf{f}(k\mathbf{x})$$

上面的式子看起来很抽象。**缩放（scale）**就是一种线性变换。例如， $\mathbf{f}(\mathbf{x}) = 2\mathbf{x}$ ，可以表示一个大小为2的统一缩放，即经过变换后矢量 $\mathbf{x}$ 的模将被放大两倍。可以发现， $\mathbf{f}(\mathbf{x}) = 2\mathbf{x}$ 是满足上面的两个条件的。同样，**旋转（rotation）**也是一种线性变换。对于线性变换来说，如果我们要对一个三维的矢量进行变换，那么仅仅使用3×3的矩阵就可以表示所有的线性变换。

线性变换除了包括旋转和缩放外，还包括**错切（shear）**、**镜像（mirroring，也被称为reflection）**、**正交投影（orthographic**

**projection**) 等，但本书着重讲述旋转和缩放变换。

但是，仅有线性变换是不够的。我们来考虑平移变换，例如  $f(x)=x+(1,2,3)$ 。这个变换就不是一个线性变换，它既不满足标量乘法，也不满足矢量加法。如果我们令  $x=(1,1,1)$ ，那么：

$$f(x)+f(x)=(4,6,8)$$

$$f(x+x)=(3,4,5)$$

可见，两个运算得到的结果是不一样的。因此，我们不能用一个  $3\times 3$  的矩阵来表示一个平移变换。这是我们不希望看到的，毕竟平移变换是非常常见的一种变换。

这样，就有了**仿射变换 (affine transform)**。仿射变换就是合并线性变换和平移变换的变换类型。仿射变换可以使用一个  $4\times 4$  的矩阵来表示，为此，我们需要把矢量扩展到四维空间下，这就是**齐次坐标空间 (homogeneous space)**。

表4.1给出了图形学中常见变换矩阵的名称和它们的特性。

**表4.1** 常见的变换种类和它们的特性 (N表示不满足该特性，Y表示满足该特性)

变换名称	是线性变换 吗	是仿射变换 吗	是可逆矩阵 吗	是正交矩阵 吗
平移矩阵	N	Y	Y	N



变换名称	是线性变换吗	是仿射变换吗	是可逆矩阵吗	是正交矩阵吗
绕坐标轴旋转的旋转矩阵	Y	Y	Y	Y
绕任意轴旋转的旋转矩阵	Y	Y	Y	Y
按坐标轴缩放的缩放矩阵	Y	Y	Y	N
错切矩阵	Y	Y	Y	N
镜像矩阵	Y	Y	Y	Y
正交投影矩阵	Y	Y	N	N
透视投影矩阵	N	N	N	N

在下面的内容中，我们将学习其中一些基本的变换类型：旋转，缩放和平移。对于正交投影和透视投影，我们将在4.6.7节中给出它们的表示方法。而对于其他变换类型，本书不再具体讨论，读者可以在本章的扩展阅读中找到更多内容。

### 4.5.2 齐次坐标

我们知道，由于 $3 \times 3$ 矩阵不能表示平移操作，我们就把其扩展到了 $4 \times 4$ 的矩阵（是的，只要多一个维度就可以实现对平移的表示）。为此，我们还需要把原来的三维矢量转换成四维矢量，也就是我们所说的**齐次坐标（homogeneous coordinate）**（事实上齐次坐标的维度可以超过四维，但本书中所说的齐次坐标将泛指四维齐次坐标）。我们可以发现，齐次坐标并没有神秘的地方，它只是为了方便计算而使用的一种表示方式而已。

如上所说，齐次坐标是一个四维矢量。那么，我们如何把三维矢量转换成齐次坐标呢？对于一个点，从三维坐标转换成齐次坐标是将其 $w$ 分量设为1，而对于方向矢量来说，需要将其 $w$ 分量设为0。这样的设置会导致，当用一个 $4 \times 4$ 矩阵对一个点进行变换时，平移、旋转、缩放都会施加于该点。但是如果是用于变换一个方向矢量，平移的效果就会被忽略。我们可以从下面的内容中理解这些差异的原因。

### 4.5.3 分解基础变换矩阵

我们已经知道，可以使用一个 $4 \times 4$ 的矩阵来表示平移、旋转和缩放。我们把表示纯平移、纯旋转和纯缩放的变换矩阵叫做基础变换矩阵。这些矩阵具有一些共同点，我们可以把一个基础变换矩阵分解成4个组成部分：

$$\begin{bmatrix} \mathbf{M}_{3 \times 3} & \mathbf{t}_{3 \times 1} \\ \mathbf{0}_{1 \times 3} & 1 \end{bmatrix}$$

其中，左上角的矩阵 $\mathbf{M}_{3 \times 3}$ 用于表示旋转和缩放， $\mathbf{t}_{3 \times 1}$ 用于表示平移， $\mathbf{0}_{1 \times 3}$ 是零矩阵，即 $\mathbf{0}_{1 \times 3} = [0 \ 0 \ 0]$ ，右下角的元素就是标量1。

接下来，我们来具体学习如何用这样一个4×4的矩阵来表示平移、旋转和缩放。

#### 4.5.4 平移矩阵

我们可以使用矩阵乘法来表示对一个点进行平移变换：

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix}$$

从结果来看我们可以很容易看出为什么这个矩阵有平移的效果：点的x、y、z分量分别增加了一个位置偏移。在3D中的可视化效果是，把点(x,y,z)在空间中平移了( $t_x, t_y, t_z$ )个单位。

有趣的是，如果我们对一个方向矢量进行平移变换，结果如下：

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}$$

可以发现，平移变换不会对方向矢量产生任何影响。这点很容易理解，我们在学习矢量的时候就说过了，矢量没有位置属性，也就是说它可以位于空间中的任意一点，因此对位置的改变（即平移）不应该对方向矢量产生影响。

现在，读者应该明白当给定一个平移操作时如何构建一个平移矩阵：基础变换矩阵中的 $\mathbf{t}_{3 \times 1}$ 矢量对应了平移矢量，左上角的矩阵 $\mathbf{M}_{3 \times 3}$ 为单位矩阵 $\mathbf{I}_3$ 。

平移矩阵的逆矩阵就是反向平移得到的矩阵，即

$$\begin{bmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & -t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

可以看出，平移矩阵并不是一个正交矩阵。

#### 4.5.5 缩放矩阵

我们可以对一个模型沿空间的x轴、y轴和z轴进行缩放。同样，我们可以使用矩阵乘法来表示一个缩放变换：

$$\begin{bmatrix} k_x & 0 & 0 & 0 \\ 0 & k_y & 0 & 0 \\ 0 & 0 & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} k_x x \\ k_y y \\ k_z z \\ 1 \end{bmatrix}$$

对方向矢量可以使用同样的矩阵进行缩放：

$$\begin{bmatrix} k_x & 0 & 0 & 0 \\ 0 & k_y & 0 & 0 \\ 0 & 0 & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} = \begin{bmatrix} k_x x \\ k_y y \\ k_z z \\ 0 \end{bmatrix}$$

如果缩放系数 $k_x=k_y=k_z$ ，我们把这样的缩放称为**统一缩放**（**uniform scale**），否则称为**非统一缩放**（**nonuniform scale**）。从外观上看，统一缩放是扩大整个模型，而非统一缩放会拉伸或挤压模型。更重要的是，统一缩放不会改变角度和比例信息，而非统一缩放会改变与模型相关的角度和比例。例如在对法线进行变换时，如果存

在非统一缩放，直接使用用于变换顶点的变换矩阵的话，就会得到错误的结果。正确的变换方法可参见4.7节。

缩放矩阵的逆矩阵是使用原缩放系数的倒数来对点或方向矢量进行缩放，即

$$\begin{bmatrix} \frac{1}{k_x} & 0 & 0 & 0 \\ 0 & \frac{1}{k_y} & 0 & 0 \\ 0 & 0 & \frac{1}{k_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

缩放矩阵一般不是正交矩阵。

上面的矩阵只适用于沿坐标轴方向进行缩放。如果我们希望在任意方向上进行缩放，就需要使用一个复合变换。其中一种方法的主要思想就是，先将缩放轴变换成标准坐标轴，然后进行沿坐标轴的缩放，再使用逆变换得到原来的缩放轴朝向。

### 4.5.6 旋转矩阵

旋转是三种常见的变换矩阵中最复杂的一种。我们知道，旋转操作需要指定一个旋转轴，这个旋转轴不一定是空间中的坐标轴，但本节所讲的旋转就是指绕着空间中的x轴、y轴或z轴进行旋转。

如果我们需要把点绕着x轴旋转 $\theta$ 度，可以使用下面的矩阵：

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

绕y轴的旋转也是类似的:

$$R_y(\theta) = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

最后，是绕z轴的旋转:

$$R_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

旋转矩阵的逆矩阵是旋转相反角度得到的变换矩阵。旋转矩阵是正交矩阵，而且多个旋转矩阵之间的串联同样是正交的。

### 4.5.7 复合变换

我们可以把平移、旋转和缩放组合起来，来形成一个复杂的变换过程。例如，可以对一个模型先进行大小为(2, 2, 2)的缩放，再绕y轴旋转30°，最后向z轴平移4个单位。复合变换可以通过矩阵的串联来实现。上面的变换过程可以使用下面的公式来计算:

$$\mathbf{P}_{new} = \mathbf{M}_{translation} \mathbf{M}_{rotation} \mathbf{M}_{scale} \mathbf{P}_{old}$$

由于上面我们使用的是列矩阵，因此阅读顺序是从右到左，即先进行缩放变换，再进行旋转变换，最后进行平移变换。需要注意的是，变换的结果是依赖于变换顺序的，由于矩阵乘法不满足交换律，因此矩阵的乘法顺序很重要。也就是说，不同的变换顺序得到的结果可能是不一样的。想象一下，如果让读者向前一步然后左转，记住此

时的位置。然后回到原位，这次先左转再向前走一步，得到的位置和上一次是不一样的。究其本质，是因为矩阵的乘法不满足交换律，因此不同的乘法顺序得到的结果是不一样的。

在绝大多数情况下，我们约定变换的顺序就是先缩放，再旋转，最后平移。

读者：为什么要约定这样的顺序，而不是其他顺序呢？

我们：因为这样的变换顺序是我们需要的。想象我们对奶牛妞妞进行一个复合变换。如果我们按先平移、再缩放的顺序进行变换，假设初始情况下妞妞位于原点，我们先按(0, 0, 5)平移它，现在它距离原点5个单位。然后再将它放大2倍，这样所有的坐标都变成了原来的2倍，而这意味着妞妞现在的位置是(0, 0, 10)，这不是我们希望的。正确的做法是，先缩放再平移。也就是说，我们先在原点对妞妞进行2倍的缩放，再进行平移，这样妞妞的大小正确了，位置也正确了。

为了从数学公式上理解变换顺序的本质，我们可以对比不同变换顺序产生的变换矩阵的表达式。如果我们只考虑对y轴的旋转的话，按先缩放、再旋转、最后平移这样的顺序组合3种变换得到的变换矩阵是：

$$\mathbf{M}_{translation}\mathbf{M}_{rotation}\mathbf{M}_{scale} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} k_x & 0 & 0 & 0 \\ 0 & k_y & 0 & 0 \\ 0 & 0 & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



$$= \begin{bmatrix} k_x \cos \theta & 0 & k_z \sin \theta & t_x \\ 0 & k_y & 0 & t_y \\ -k_x \sin \theta & 0 & k_z \cos \theta & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

而如果我们使用了其他变换顺序，例如先平移，再缩放，最后旋转，那么得到的变换矩阵是：

$$\begin{aligned} \mathbf{M}_{rotation} \mathbf{M}_{scale} \mathbf{M}_{translation} &= \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} k_x 0 & 0 & 0 & 0 \\ 0 & k_y & 0 & 0 \\ 0 & 0 & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} k_x \cos \theta & 0 & k_z \sin \theta & t_x k_x \cos \theta + t_z k_z \sin \theta \\ 0 & k_y & 0 & t_y k_y \\ -k_x \sin \theta & 0 & k_z \cos \theta & -t_x k_x \sin \theta + t_z k_z \cos \theta \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

从两个结果可以看出，得到的变换矩阵是不一样的。

除了需要注意不同类型的变换顺序外，我们有时还需要小心旋转的变换顺序。在4.5.6节中，我们给出了分别绕x轴、y轴和z轴旋转的变换矩阵。一个问题是，如果我们需要同时绕着3个轴进行旋转，是先绕x轴、再绕y轴最后绕z轴旋转还是按其他的旋转顺序呢？

当我们直接给出 $(\theta_x, \theta_y, \theta_z)$ 这样的旋转角度时，需要定义一个旋转顺序。在Unity中，这个旋转顺序是zxy，这在旋转相关的API文档中都有说明。这意味着，当给定 $(\theta_x, \theta_y, \theta_z)$ 这样的旋转角度时，得到的组合旋转变换矩阵是：

$$[\mathbf{M}_{rotate_z} \mathbf{M}_{rotate_x} \mathbf{M}_{rotate_y}] = \begin{bmatrix} \cos \theta_z & -\sin \theta_z & 0 & 0 \\ \sin \theta_z & \cos \theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta_x & -\sin \theta_x & 0 \\ 0 & \sin \theta_x & \cos \theta_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta_y & 0 & \sin \theta_y & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta_y & 0 & \cos \theta_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

一些读者会有疑问：上面的公式书写顺序是不是反了？不是说列矩阵要从右往左读吗？这样一来顺序不就颠倒了吗？实际上，有一个非常重要的东西我们没有说明白，那就是旋转时使用的坐标系。给定一个旋转顺序（例如这里的 $zxy$ ），以及它们对应的旋转角度 $(\theta_x, \theta_y, \theta_z)$ ，有两种坐标系可以选择。

- 绕坐标系E下的 $z$ 轴旋转 $\theta_z$ ，绕坐标系E下的 $y$ 轴旋转 $\theta_y$ ，绕坐标系E下的 $x$ 轴旋转 $\theta_x$ ，即进行一次旋转时不一起旋转当前坐标系。
- 绕坐标系E下的 $z$ 轴旋转 $\theta_z$ ，在坐标系E下绕 $z$ 轴旋转 $\theta_z$ 后的新坐标系E'下的 $y$ 轴旋转 $\theta_y$ ，在坐标系E'下绕 $y$ 轴旋转 $\theta_y$ 后的新坐标系E''下的 $x$ 轴旋转 $\theta_x$ ，即在旋转时，把坐标系一起转动。

很容易知道，这两种选择的结果是不一样的。但如果把它们旋转顺序颠倒一下，它们得到的结果就会是一样的！说得明白点，在第一种情况下，按 $zxy$ 顺序旋转和在第二种情况下，按 $yxz$ 顺序旋转是一样的。而Unity文档中说明的旋转顺序指的是在第一种情况下的顺序。

和上面不同类型的变换顺序导致的问题类似，不同的旋转顺序得到的结果也可能是不一样的。我们同样可以通过对比不同旋转顺序得到的变换矩阵来理解为什么会出现这样的不同。而这个验证过程留给读者作为练习。

## 4.6 坐标空间

我们已经学会了如何使用矩阵来表示基本的变换，如平移、旋转和缩放。而在本节中，我们将关注如何使用这些变换来对坐标空间进行变换。

我们在第2章渲染流水线中就接触了坐标空间的变换。例如，在学习顶点着色器流水线阶段时，我们说过，顶点着色器最基本的功能就是把模型的顶点坐标从模型空间转换到齐次裁剪坐标空间中。

渲染游戏的过程可以理解成是把一个个顶点经过层层处理最终转化到屏幕上的过程，那么本节我们就将学习这个转换的过程是如何实现的。更具体来说，顶点是经过了哪些坐标空间后，最后被画在了我们的屏幕上。

### 4.6.1 为什么要使用这么多不同的坐标空间

我们先要回答读者的一个疑问。在编写Shader的过程中，很多看起来很难理解和复杂的数学运算都是为了在不同坐标空间之间转换点和矢量。看起来，这么多的坐标空间就是“万恶之源”啊！很多人都有这样的疑问：“为什么我们不能只使用一个坐标空间来做所有的事情呢？这样一来我们不就不用学习这些烦人的数学公式了吗？这样世界将变得多美好啊！”

事情看起来虽然是这样——在只有一个坐标空间的世界里，Shader的开发者会生活得更加美好。但事实是，一旦你真的这么做了，就会发现理想和现实之间的差距：我们不可以也不愿意抛弃这些不同的坐标空间。

事实上，在我们的生活中，我们也总是使用不同的坐标空间来交流。现在正在读这本书的你，很可能正坐在办公室或书房中。如果问你：“办公室的饮水机在哪里？”你大概会回答：“在办公室门的左方3米处。”这里，你很自然地使用了以门为原点的坐标空间。现在，公司的前台小姐走进门来，你非常惊讶地看到她脸上还残留有中午吃饭的米粒！我们假设正在读这本书的你是一个好心而且不喜欢看别人笑话的人，这时你可能会提醒她：“嘿，你左脸上面有些东西没有擦掉！”此时，你又使用了以前台小姐的嘴巴为原点的坐标空间。如果只有一个坐标系会怎么样呢？你可以尝试一下使用以你的办公室的门为原点的坐标空间来描述前台小姐脸上的一粒饭粒。

再比如，我们每个人所生活的城市可以看成是一个世界坐标系（三维渲染里的世界坐标系将在4.6.5节中讲到），这个坐标系的坐标轴可以认为是由东南西北这些定义的方向轴。如果一个陌生人向你问路，你很有可能会说：“向东走800米上桥，然后再向南走50米就到了”。但是我们知道，现实生活中有很多人是不分清东南西北的（在作者小时候，经常使用“上北下南左西右东”来傻傻地判断东南西北，因此总是得到错误方位）。如果现在有一个饥肠辘辘又分清东南西北的路人来问你最近的餐厅怎么走，你可能会说：“你先往前走50米，到了路口向左拐100米就有一家非常好吃的烤鸭店。”此时，你使用的是以这个路人为原点的坐标空间。想象一下，如果在这个世界上我们只能使用东南西北来描述所有东西的话，该会有多少人会被饿死。

由此可见，我们需要在不同的情况下使用不同的坐标空间，因为一些概念只有在特定的坐标空间下才有意义，才更容易理解。这也是为什么在渲染中我们要使用这么多坐标空间。

在开始介绍一些不同的坐标空间之前，读者需要注意，所有的坐标空间在理论上都是平等的，没有谁优谁劣之分，不会因为我们从一个坐标空间转换到另一个坐标空间计算就出错了。但是，在特定的情况下，一些坐标空间的确比另一些坐标空间更加吸引人。

现在，就让我们来看一下在游戏渲染流水线中，一个顶点到底经过了怎样的空间变换。

## 4.6.2 坐标空间的变换

我们先要为后面的内容做些数学铺垫。在渲染流水线中，我们往往需要把一个点或方向矢量从一个坐标空间转换到另一个坐标空间。这个过程到底是怎么实现的呢？

我们把问题一般化。我们知道，要想定义一个坐标空间，必须指明其原点位置和3个坐标轴的方向。而这些数值实际上是相对于另一个坐标空间的（读者需要记住，所有的都是相对的）。也就是说，坐标空间会形成一个层次结构——每个坐标空间都是另一个坐标空间的子空间，反过来说，每个空间都有一个父（parent）坐标空间。对坐标空间的变换实际上就是在父空间和子空间之间对点和矢量进行变换。

假设，现在有父坐标空间 $\mathbf{P}$ 以及一个子坐标空间 $\mathbf{C}$ 。我们知道在父坐标空间中子坐标空间的原点位置以及3个单位坐标轴。我们一般会有两种需求：一种需求是把子坐标空间下表示的点或矢量 $\mathbf{A}_c$ 转换到父坐标空间下的表示 $\mathbf{A}_p$ ，另一个需求是反过来，即把父坐标空间下表示的点或矢量 $\mathbf{B}_p$ 转换到子坐标空间下的表示 $\mathbf{B}_c$ 。我们可以使用下面的公式来表示这两种需求：

$$\mathbf{A}_p = \mathbf{M}_{c \rightarrow p} \mathbf{A}_c$$

$$\mathbf{B}_c = \mathbf{M}_{p \rightarrow c} \mathbf{B}_p$$

其中， $\mathbf{M}_{c \rightarrow p}$ 表示的是从子坐标空间变换到父坐标空间的变换矩阵，而 $\mathbf{M}_{p \rightarrow c}$ 是其逆矩阵（即反向变换）。那么，现在的问题就是，如何求解这些变换矩阵？事实上，我们只需要解出两者之一即可，另一个矩阵可以通过求逆矩阵的方式来得到。

下面，我们就来讲解如何求出从子坐标空间到父坐标空间的变换矩阵 $\mathbf{M}_{c \rightarrow p}$ 。

首先，我们来回顾一个看似很简单的问题：当给定一个坐标空间以及其中一点 $(a,b,c)$ 时，我们是如何知道该点的位置的呢？我们可以通过4个步骤来确定它的位置：

- (1) 从坐标空间的原点开始；
- (2) 向x轴方向移动 $a$ 个单位；
- (3) 向y轴方向移动 $b$ 个单位；
- (4) 向z轴方向移动 $c$ 个单位。

需要说明的是，上面的步骤只是我们的想象，这个点实际上并没有发生移动。上面的步骤看起来再简单不过了，坐标空间的变换就蕴含在上面的4个步骤中。现在，我们已知子坐标空间 $\mathbf{C}$ 的3个坐标轴在父坐标空间 $\mathbf{P}$ 下的表示 $\mathbf{x}_c$ 、 $\mathbf{y}_c$ 、 $\mathbf{z}_c$ ，以及其原点位置 $\mathbf{O}_c$ 。当给定一个子坐

标空间中的一点  $\mathbf{A}_c = (a, b, c)$ ，我们同样可以依照上面4个步骤来确定其在父坐标空间下的位置  $\mathbf{A}_p$ ：

### 1. 从坐标空间的原点开始

这很简单，我们已经知道了子坐标空间的原点位置  $\mathbf{O}_c$ 。

### 2. 向x轴方向移动a个单位

仍然很简单，因为我们已经知道了x轴的矢量表示，因此可以得到

$$\mathbf{O}_c + a\mathbf{x}_c$$

### 3. 向y轴方向移动b个单位

同样的道理，这一步就是：

$$\mathbf{O}_c + a\mathbf{x}_c + b\mathbf{y}_c$$

### 4. 向z轴方向移动c个单位

最后，就可以得到

$$\mathbf{O}_c + a\mathbf{x}_c + b\mathbf{y}_c + c\mathbf{z}_c$$

现在，我们已经求出了  $\mathbf{M}_{c \rightarrow p}$ ！什么？你没看出来吗？我们再来看一下最后得到的式子：

$$\mathbf{A}_p = \mathbf{O}_c + a\mathbf{x}_c + b\mathbf{y}_c + c\mathbf{z}_c$$

读者可能会问，这个式子里根本没有矩阵啊！其实我们只要稍稍使用一点“魔法”，矩阵就会出现在上面的式子中：



$$\begin{aligned}
\mathbf{A}_p &= \mathbf{O}_c + a\mathbf{x}_c + b\mathbf{y}_c + c\mathbf{z}_c \\
&= (x_{O_c}, y_{O_c}, z_{O_c}) + a(x_{x_c}, y_{x_c}, z_{x_c}) + b(x_{y_c}, y_{y_c}, z_{y_c}) + c(x_{z_c}, y_{z_c}, z_{z_c}) \\
&= (x_{O_c}, y_{O_c}, z_{O_c}) + \begin{bmatrix} x_{x_c} & x_{y_c} & x_{z_c} \\ y_{x_c} & y_{y_c} & y_{z_c} \\ z_{x_c} & z_{y_c} & z_{z_c} \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} \\
&= (x_{O_c}, y_{O_c}, z_{O_c}) + \begin{bmatrix} | & | & | \\ x_c & y_c & z_c \\ | & | & | \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix}
\end{aligned}$$

其中“|”符号表示是按列展开的。上面的式子实际上就是使用了我们之前所学的公式而已。但这个最后的表达式还不是很漂亮，因为还存在加法表达式，即平移变换。我们已经知道3×3的矩阵无法表示平移变换，因此为了得到一个更漂亮的结果，我们把上面的式子扩展到齐次坐标空间中，得

$$\begin{aligned}
\mathbf{A}_p &= (x_{O_c}, y_{O_c}, z_{O_c}, 1) + \begin{bmatrix} | & | & | & 0 \\ \mathbf{x}_c & \mathbf{y}_c & \mathbf{z}_c & 0 \\ | & | & | & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 & x_{O_c} \\ 0 & 1 & 0 & y_{O_c} \\ 0 & 0 & 1 & z_{O_c} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} | & | & | & 0 \\ \mathbf{x}_c & \mathbf{y}_c & \mathbf{z}_c & 0 \\ | & | & | & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} | & | & | & x_{O_c} \\ \mathbf{x}_c & \mathbf{y}_c & \mathbf{z}_c & y_{O_c} \\ | & | & | & z_{O_c} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} | & | & | & | \\ \mathbf{x}_c & \mathbf{y}_c & \mathbf{z}_c & O_c \\ | & | & | & | \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix}
\end{aligned}$$

那么现在，你看到 $\mathbf{M}_{c \rightarrow p}$ 在哪里了吧？没错，

$$\mathbf{M}_{c \rightarrow p} = \begin{bmatrix} | & | & | & | \\ \mathbf{x}_c & \mathbf{y}_c & \mathbf{z}_c & \mathbf{O}_c \\ | & | & | & | \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

读者：这个看起来太神奇了！怎么就变着变着就出现了矩阵呢？

我们：上面只是运用了一些基础的矢量和矩阵运算，一旦当你真正理解了这些运算就会发现上面的过程只是简单地推导了一下而已。

一旦求出来 $\mathbf{M}_{c \rightarrow p}$ ， $\mathbf{M}_{p \rightarrow c}$ 就可以通过求逆矩阵的方式求出来，因为从坐标空间 $\mathbf{C}$ 变换到坐标空间 $\mathbf{P}$ 与从坐标空间 $\mathbf{P}$ 变换到坐标空间 $\mathbf{C}$ 是互逆的两个过程。

可以看出来，变换矩阵 $\mathbf{M}_{c \rightarrow p}$ 实际上可以通过坐标空间 $\mathbf{C}$ 在坐标空间 $\mathbf{P}$ 中的原点和坐标轴的矢量表示来构建出来：把3个坐标轴依次放入矩阵的前3列，把原点矢量放到最后一列，再用0和1填充最后一行即可。

需要注意的是，这里我们并没有要求3个坐标轴 $\mathbf{x}_c$ 、 $\mathbf{y}_c$ 和 $\mathbf{z}_c$ 是单位矢量，事实上，如果存在缩放的话，这3个矢量值很可能不是单位矢量。

更加令人振奋的是，我们可以利用反向思维，从这个变换矩阵反推来获取子坐标空间的原点和坐标轴方向！例如，当我们已知从模型空间到世界空间的一个 $4 \times 4$ 的变换矩阵，可以提取它的第一列再进行归一化后（为了消除缩放的影响）来得到模型空间的x轴在世界空间下的单位矢量表示。同样的方法可以提取y轴和z轴。我们可以从另一个角度来理解这个提取过程。因为矩阵 $\mathbf{M}_{c \rightarrow p}$ 可以把一个方向矢量从坐标空间

**C**变换到坐标空间**P**中，那么，我们只需要用它来变换坐标空间**C**中的x轴(1,0,0,0)，即使用矩阵乘法 $\mathbf{M}_{c \rightarrow p}[1 \ 0 \ 0 \ 0]^T$ ，得到的结果正是 $\mathbf{M}_{c \rightarrow p}$ 的第一列。

另一个有趣的情况是，对方向矢量的坐标空间变换。我们知道，矢量是没有位置的，因此坐标空间的原点变换是可以忽略的。也就是说，我们仅仅平移坐标系的原点是不会对矢量造成任何影响的。那么，对矢量的坐标空间变换就可以使用3×3的矩阵来表示，因为我们不需要表示平移变换。那么变换矩阵就是：

$$\mathbf{M}_{c \rightarrow p} = \begin{bmatrix} | & | & | \\ \mathbf{x}_c & \mathbf{y}_c & \mathbf{z}_c \\ | & | & | \end{bmatrix}$$

在Shader中，我们常常会看到截取变换矩阵的前3行前3列来对法线方向、光照方向来进行空间变换，这正是原因所在。

现在，我们再来关注 $\mathbf{M}_{p \rightarrow c}$ 。我们前面讲到，可以通过求 $\mathbf{M}_{c \rightarrow p}$ 的逆矩阵的方式求解出来反向变换 $\mathbf{M}_{p \rightarrow c}$ 。但有一种情况我们不需求解逆矩阵就可以得到 $\mathbf{M}_{p \rightarrow c}$ ，这种情况就是 $\mathbf{M}_{c \rightarrow p}$ 是一个正交矩阵。如果它是一个正交矩阵的话， $\mathbf{M}_{c \rightarrow p}$ 的逆矩阵就等于它的转置矩阵。这意味着我们不需要进行复杂的求逆操作就可以得到反向变换。也就是说，

$$\begin{aligned} \mathbf{M}_{p \rightarrow c} &= \begin{bmatrix} | & | & | \\ \mathbf{x}_p & \mathbf{y}_p & \mathbf{z}_p \\ | & | & | \end{bmatrix} = \mathbf{M}_{c \rightarrow p}^{-1} = \mathbf{M}_{c \rightarrow p}^T \\ &= \begin{bmatrix} - & \mathbf{x}_c & - \\ - & \mathbf{y}_c & - \\ - & \mathbf{z}_c & - \end{bmatrix} \end{aligned}$$

而现在，我们不仅可以根据变换矩阵 $\mathbf{M}_{C \rightarrow P}$ 反推出子坐标空间的坐标轴方向在父坐标空间中的表示 $\mathbf{x}_C$ 、 $\mathbf{y}_C$ 和 $\mathbf{z}_C$ ，还可以反推出父坐标空间的坐标轴方向在子坐标空间中的表示 $\mathbf{x}_P$ 、 $\mathbf{y}_P$ 和 $\mathbf{z}_P$ ，这些坐标轴对应的就是 $\mathbf{M}_{C \rightarrow P}$ 的每一行！也就是说，如果我们知道坐标空间变换矩阵 $\mathbf{M}_{A \rightarrow B}$ 是一个正交矩阵，那么我们可以提取它的第一列来得到坐标空间 $\mathbf{A}$ 的 $x$ 轴在坐标空间 $\mathbf{B}$ 下的表示，还可以提取它的第一行来得到坐标空间 $\mathbf{B}$ 的 $x$ 轴在坐标空间 $\mathbf{A}$ 下的表示。反过来，如果我们知道坐标空间 $\mathbf{B}$ 的 $x$ 轴、 $y$ 轴和 $z$ 轴（必须是单位矢量，否则构建出来的就不是正交矩阵了）在坐标空间 $\mathbf{A}$ 下的表示，就可以把它们依次放在矩阵的每一行就可以得到从 $\mathbf{A}$ 到 $\mathbf{B}$ 的变换矩阵了。

读者：天呐，我的脑子已经完全乱掉了，一会儿从 $\mathbf{P}$ 到 $\mathbf{C}$ ，一会儿又从 $\mathbf{C}$ 到 $\mathbf{P}$ ，一会儿是行，一会儿又是列，我自己写的时候一定会搞不清楚！

我们：我们知道这个过程很容易造成思维的混乱，因此才要花费大量的篇幅来解释背后的数学原理。只有知道了这些原理，遇到疑问时你才知道怎样去验证结果的正确性。例如像下面这样。

当你不知道把坐标轴的表示是按行放还是按列放的时候，不妨先选择一种摆放方式来得到变换矩阵。例如，现在我们把一个矢量从坐标空间 $\mathbf{A}$ 变换到坐标空间 $\mathbf{B}$ ，而且我们已经知道坐标空间 $\mathbf{B}$ 的 $x$ 轴、 $y$ 轴、 $z$ 轴在空间 $\mathbf{A}$ 下的表示，即 $\mathbf{x}_B$ 、 $\mathbf{y}_B$ 和 $\mathbf{z}_B$ 。那么想要得到从 $\mathbf{A}$ 到 $\mathbf{B}$ 的变换矩阵 $\mathbf{M}_{A \rightarrow B}$ ，我们是把它们按列放呢还是按行放呢？如果读者实在想不起来正确答案，我们不妨先随便选择一种方式，例如按列摆放。那么，

$$\mathbf{M}_{A \rightarrow B} = \begin{bmatrix} | & | & | \\ \mathbf{x}_B & \mathbf{y}_B & \mathbf{z}_B \\ | & | & | \end{bmatrix}, \text{ 注意, 这个矩阵是不对的}$$

现在, 我们可以非常快速地来验证它是否是正确的。方法就是, 用 $\mathbf{M}_{A \rightarrow B}$ 来变换 $\mathbf{x}_B$ 。在计算前我们先想一下这个结果, 如果我们用变换矩阵来变换 $\mathbf{B}$ 的 $\mathbf{x}$ 轴的话, 那么结果应该是 $(1,0,0)$ 才对。因为当变换到空间 $\mathbf{B}$ 中时,  $\mathbf{x}$ 轴的指向就是 $(1,0,0)$ 。好了, 我们可以来进行真正的计算来验证它了:

$$\mathbf{M}_{A \rightarrow B} \mathbf{x}_B = \begin{bmatrix} | & | & | \\ \mathbf{x}_B & \mathbf{y}_B & \mathbf{z}_B \\ | & | & | \end{bmatrix} \mathbf{x}_B$$

读者看到这里会有疑问, “我不知道这个结果是什么啊”。没错, 这不是你的计算有问题, 而是上式的计算结果的确不可知。这种时候你就会发现我们的摆放方式选择错了。现在, 我们使用正确的摆放方式, 即按行来摆放, 那么就有:

$$\mathbf{M}_{A \rightarrow B} \mathbf{x}_B = \begin{bmatrix} - & \mathbf{x}_B & - \\ - & \mathbf{y}_B & - \\ - & \mathbf{z}_B & - \end{bmatrix} \mathbf{x}_B = \begin{bmatrix} \mathbf{x}_B \cdot \mathbf{x}_B \\ \mathbf{y}_B \cdot \mathbf{x}_B \\ \mathbf{z}_B \cdot \mathbf{x}_B \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

这次结果就和我们预期的一样了。

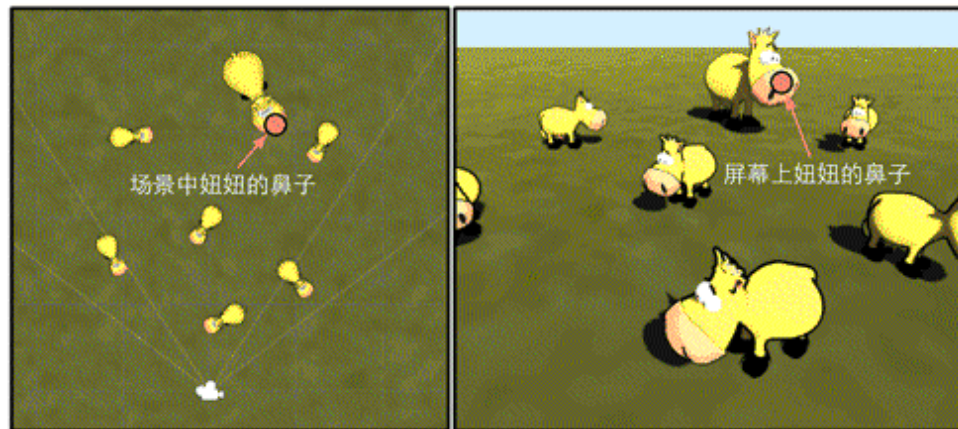
理解上面的原理和过程非常重要。我们在本书的后面也会经常遇到坐标空间的变换。

### 4.6.3 顶点的坐标空间变换过程

我们知道，在渲染流水线中，一个顶点要经过多个坐标空间的变换才能最终被画在屏幕上。一个顶点最开始是在模型空间（见4.6.4节）中定义的，最后它将会变换到屏幕空间（见4.6.8节）中，得到真正的屏幕像素坐标。因此，接下来的内容我们将解释顶点要进行的各种空间变换的过程。

为了帮助读者理解这个过程，我们将建立在农场游戏的实例背景下，每讲到一种空间变换，我们会解释如何应用到这个案例中。

在我们的农场游戏中，妞妞很好奇自己是如何被渲染到屏幕上的。它只知道自己和一群小伙伴在农场里快乐地吃草，而前面有一个摄像机一直在观察它们，如图4.31所示。妞妞特别喜欢自己的鼻子，它想知道鼻子是怎么被画到屏幕上的？



▲图4.31 场景中的妞妞（左图）和屏幕上的妞妞（右图）。妞妞想知道，自己的鼻子是如何被画到屏幕上的

在下面的内容中，我们将了解妞妞的鼻子是如何一步步画到屏幕上的。

#### 4.6.4 模型空间

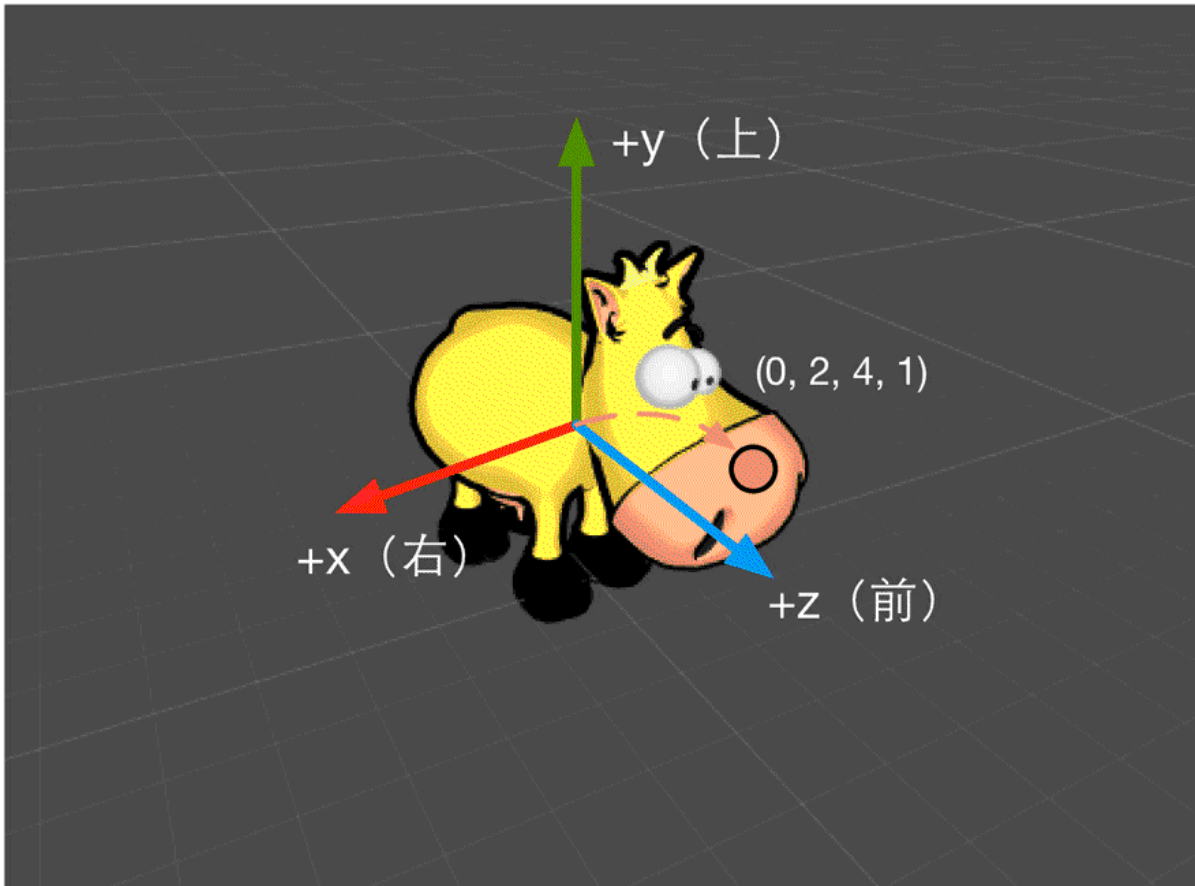
**模型空间 (model space)**，如它的名字所暗示的那样，是和某个模型或者说是对象有关的。有时模型空间也被称为**对象空间 (object space)**或**局部空间 (local space)**。每个模型都有自己独立的坐标空间，当它移动或旋转的时候，模型空间也会跟着它移动和旋转。把我们自己当成游戏中的模型的话，当我们在办公室里移动时，我们的模型空间也在跟着移动，当我们转身时，我们本身的前后左右方向也在跟着改变。

在模型空间中，我们经常使用一些方向概念，例如“前 (forward)”“后 (back)”“左 (left)”、“右 (right)”、“上 (up)”、“下 (down)”。在本书中，我们把这些方向称为自然方向。模型空间中的坐标轴通常会使用这些自然方向。在4.2.4节中我们讲过，Unity在模型空间中使用的是左手坐标系，因此在模型空间中，+x轴、+y轴、+z轴分别对应的是模型的右、上和前向。需要注意的是，模型坐标空间中的x轴、y轴、z轴和自然方向的对应不一定是上述这种关系，但由于Unity使用的是这样的约定，因此本书将使用这种方式。我们可以在Hierarchy视图中单击任意对象就可以看见它们对应的模型空间的3个坐标轴。

模型空间的原点和坐标轴通常是由美术人员在建模软件里确定好的。当导入到Unity中后，我们可以在顶点着色器中访问到模型的顶点信息，其中包含了每个顶点的坐标。这些坐标都是相对于模型空间中的原点（通常位于模型的重心）定义的。



当我们把妞妞放到场景中时，就会有一个模型坐标空间时刻跟着它。妞妞鼻子的位置可以通过访问顶点属性来得到。假设这个位置是 $(0, 2, 4)$ ，由于顶点变换中往往包含了平移变换，因此需要把其扩展到齐次坐标系下，得到顶点坐标是 $(0, 2, 4, 1)$ ，如图4.32所示。



▲图4.32 在我们的农场游戏中，每个奶牛都有自己的模型坐标系。在模型坐标系中妞妞鼻子的位置是 $(0, 2, 4, 1)$

### 4.6.5 世界空间

**世界空间 (world space)** 是一个特殊的坐标系，因为它建立了我们所关心的最大的空间。一些读者可能会指出，空间可以是无限大

的，怎么会有“最大”这一说呢？这里说的最大指的是一个宏观的概念，也就是说它是我们所关心的最外层的坐标空间。以我们的农场游戏为例，在这个游戏里世界空间指的就是农场，我们不关心这个农场是在什么地方，在这个虚拟的游戏世界里，农场就是最大的空间概念。

世界空间可以被用于描述绝对位置（较真的读者可能会再一次提醒我，没有绝对的位置。没错，但我相信读者可以明白这里绝对的意思）。在本书中，绝对位置指的就是在世界坐标系中的位置。通常，我们会把世界空间的原点放置在游戏空间的中心。

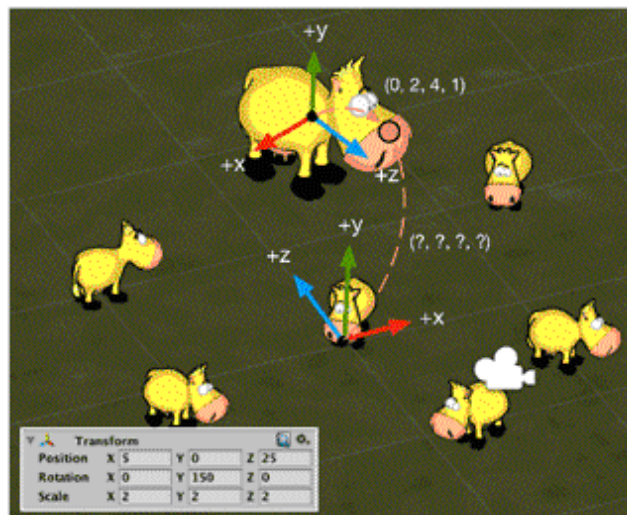
在Unity中，世界空间同样使用了左手坐标系。但它的x轴、y轴、z轴是固定不变的。在Unity中，我们可以通过调整Transform组件中的Position属性来改变模型的位置，这里的位置指的是相对于这个Transform的父节点（parent）的模型坐标空间中的原点定义的。如果一个Transform没有任何父节点，那么这个位置就是在世界坐标系中的位置，如图4.33所示。我们可以想象成还有一个虚拟的根模型，这个根模型的模型空间就是世界空间，所有的游戏对象都附属于此根模型。同样，Transform中的Rotation和Scale也是同样的道理。

顶点变换的第一步，就是将顶点坐标从模型空间变换到世界空间中。这个变换通常叫做**模型变换（model transform）**。

现在，我们来对妞妞的鼻子进行模型变换。为此，我们首先需要知道妞妞在世界坐标系中进行了哪些变换，这可以通过面板中的Transform组件来得到相关的变换信息，如图4.34所示。



▲图4.33 Unity的Transform组件可以调节模型的位置。如果Transform有父节点，如图中的“Mesh”，那么Position将是在其父节点（这里是“Cow”）的模型空间中的位置；如果没有父节点，Position就是在世界空间中的位置



▲图4.34 农场游戏中的世界空间。世界空间的原点被放置在农场的中心。左下角显示了妞妞在世界空间中所做的变换。我们想要把妞妞的鼻子从模型空间变换到世界空间中

根据Transform组件上的信息，我们知道在世界空间中，妞妞进行了(2, 2, 2)的缩放，又进行了(0, 150, 0)的旋转以及(5, 0, 25)的平移。注

意这里的变换顺序是不能互换的，即先进行缩放，再进行旋转，最后是平移。据此我们可以构建出模型变换的变换矩阵：

$$\begin{aligned}
 \mathbf{M}_{model} &= \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} k_x & 0 & 0 & 0 \\ 0 & k_y & 0 & 0 \\ 0 & 0 & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 25 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -0.866 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \\ -0.5 & 0 & -0.866 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} -1.732 & 0 & 1 & 5 \\ 0 & 2 & 0 & 0 \\ -1 & 0 & -1.732 & 25 \\ 0 & 0 & 0 & 1 \end{bmatrix}
 \end{aligned}$$

现在我们可以用它来对妞妞的鼻子进行模型变换了：

$$\begin{aligned}
 \mathbf{P}_{world} &= \mathbf{M}_{model} \mathbf{P}_{model} \\
 &= \begin{bmatrix} -1.732 & 0 & 1 & 5 \\ 0 & 2 & 0 & 0 \\ -1 & 0 & -1.732 & 25 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 4 \\ 1 \end{bmatrix} = \begin{bmatrix} 9 \\ 4 \\ 18.072 \\ 1 \end{bmatrix}
 \end{aligned}$$

也就是说，在世界空间下，妞妞鼻子的位置是(9, 4, 18.072)。注意，这里的浮点数都是近似值，这里近似到小数点后3位。实际数值和Unity采用的浮点值精度有关。

#### 4.6.6 观察空间

**观察空间（view space）**也被称为**摄像机空间（camera space）**。观察空间可以认为是模型空间的一个特例——在所有的模型中有一个非常特殊的模型，即摄像机（虽然通常来说摄像机本身是不可见的），它的模型空间值得我们单独拿出来讨论，也就是观察空间。

摄像机决定了我们渲染游戏所使用的视角。在观察空间中，摄像机位于原点，同样，其坐标轴的选择可以是任意的，但由于本书讨论的是以Unity为主，而Unity中观察空间的坐标轴选择是：**+x轴**指向右方，**+y轴**指向上方，而**+z轴**指向的是摄像机的后方。读者在这里可能觉得很奇怪，我们之前讨论的模型空间和世界空间中**+z轴**指的都是物体的前方，为什么这里不一样了呢？这是因为，Unity在模型空间和世界空间中选用的都是左手坐标系，而在观察空间中使用的是右手坐标系。这是符合OpenGL传统的，在这样的观察空间中，摄像机的正前方指向的是**-z轴**方向。

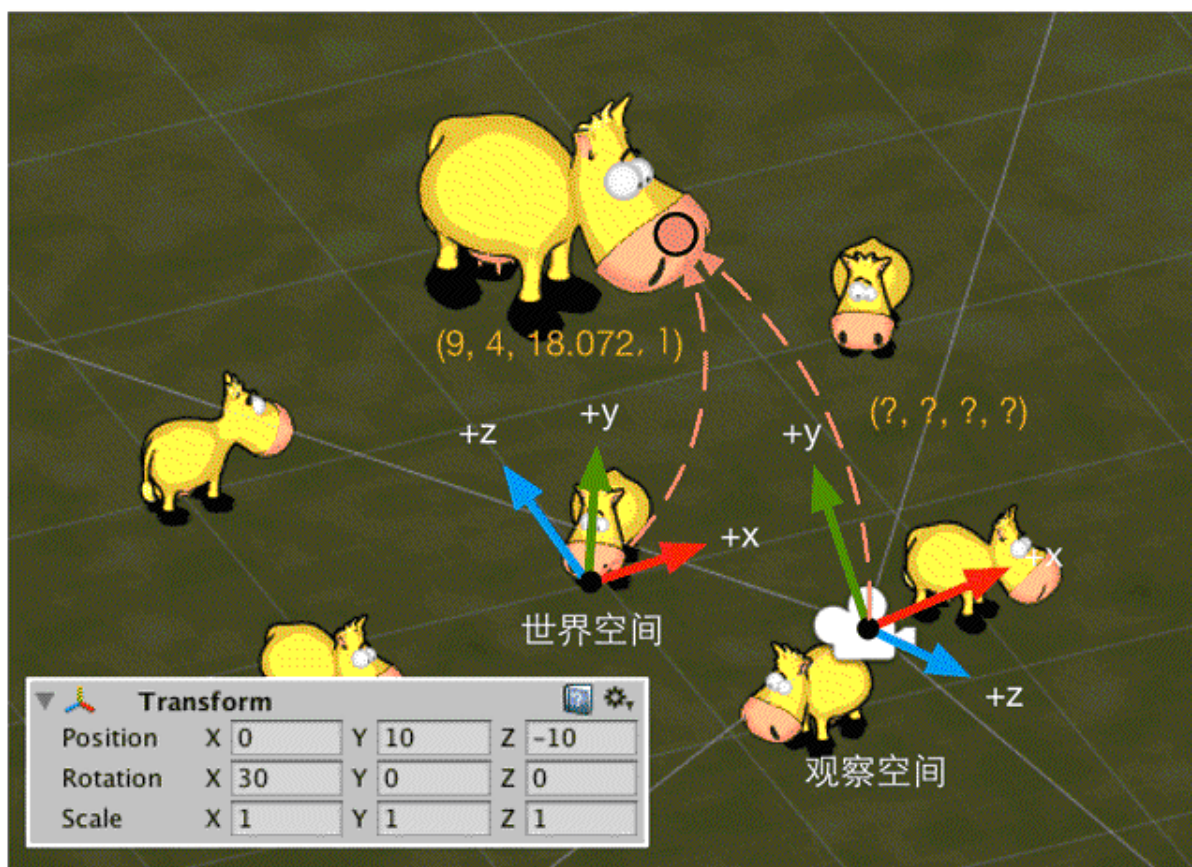
这种左右手坐标系之间的改变很少会对我们在Unity中的编程产生影响，因为Unity为我们做了很多渲染的底层工作，包括很多坐标空间的转换。但是，如果读者需要调用类似**Camera.cameraToWorldMatrix**、**Camera.worldToCameraMatrix**等接口自行计算某模型在观察空间中的位置，就要小心这样的差异。

最后要提醒读者的一点是，观察空间和屏幕空间（详见4.6.8节）是不同的。观察空间是一个三维空间，而屏幕空间是一个二维空间。从观察空间到屏幕空间的转换需要经过一个操作，那就是**投影（projection）**。我们后面就会讲到。



顶点变换的第二步，就是将顶点坐标从世界空间变换到观察空间中。这个变换通常叫做**观察变换（view transform）**。

回到我们的农场游戏。现在我们需要把妞妞的鼻子从世界空间变换到观察空间中。为此，我们需要知道世界坐标系下摄像机的变换信息。这同样可以通过摄像机面板中的Transform组件得到，如图4.35所示。



▲图4.35 农场游戏中摄像机的观察空间。观察空间的原点位于摄像机处。注意在观察空间中，摄像机的前向是z轴的负方向（图中只画出了z轴正方向），这是因为Unity在观察空间中使用了右手坐标系。左下角显示了摄像机在世界空间中所做的变换。我们想要把妞妞的鼻子从世界空间变换到观察空间中

为了得到顶点在观察空间中的位置，我们可以有两种方法。一种方法是计算观察空间的三个坐标轴在世界空间下的表示，然后根据4.6.2节中讲到的方法，构建出从观察空间变换到世界空间的变换矩阵，再对该矩阵求逆来得到从世界空间变换到观察空间的变换矩阵。我们还可以使用另一种方法，即想象平移整个观察空间，让摄像机原点位于世界坐标的原点，坐标轴与世界空间中的坐标轴重合即可。这两种方法得到的变换矩阵都是一样的，不同的只是我们思考的方式。

这里我们使用第二种方法。由Transform组件可以知道，摄像机在世界空间中的变换是先按(30, 0, 0)进行旋转，然后按(0, 10, -10)进行了平移。那么，为了把摄像机重新移回到初始状态（这里指摄像机原点位于世界坐标的原点、坐标轴与世界空间中的坐标轴重合），我们需要进行逆向变换，即先按(0, -10, 10)平移，以便将摄像机移回到原点，再按(-30, 0, 0)进行旋转，以便让坐标轴重合。因此，变换矩阵就是：

$$\begin{aligned}
 M_{view} &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.866 & 0.5 & 0 \\ 0 & -0.5 & 0.866 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -10 \\ 0 & 0 & 1 & 10 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.866 & 0.5 & -3.66 \\ 0 & -0.5 & 0.866 & 13.66 \\ 0 & 0 & 0 & 1 \end{bmatrix}
 \end{aligned}$$



但是，由于观察空间使用的是右手坐标系，因此需要对z分量进行取反操作。我们可以通过乘以另一个特殊的矩阵来得到最终的观察变换矩阵：

$$\begin{aligned}
 \mathbf{M}_{view} &= \mathbf{M}_{negate\ z} \mathbf{M}_{view} \\
 &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.866 & 0.5 & -3.66 \\ 0 & -0.5 & 0.866 & 13.66 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.866 & 0.5 & -3.66 \\ 0 & 0.5 & -0.866 & -13.66 \\ 0 & 0 & 0 & 1 \end{bmatrix}
 \end{aligned}$$

现在我们可以用它来对妞妞的鼻子进行顶点变换了：

$$\begin{aligned}
 \mathbf{P}_{view} &= \mathbf{M}_{view} \mathbf{P}_{world} \\
 &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.866 & 0.5 & -3.66 \\ 0 & 0.5 & -0.866 & -13.66 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 9 \\ 4 \\ 18.072 \\ 1 \end{bmatrix} = \begin{bmatrix} 9 \\ 8.84 \\ -27.31 \\ 1 \end{bmatrix}
 \end{aligned}$$

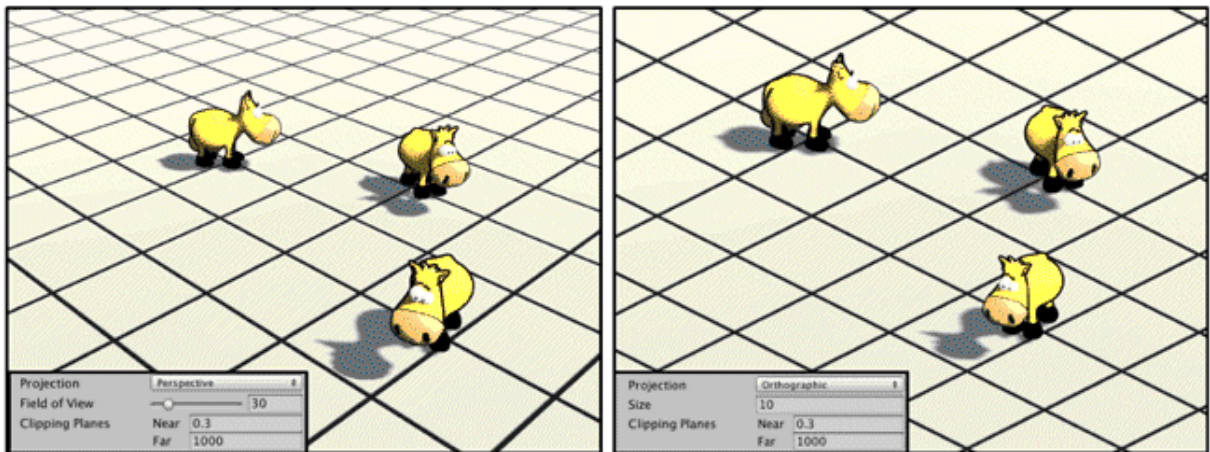
这样，我们就得到了观察空间中妞妞鼻子的位置——(9, 8.84, -27.31)。

#### 4.6.7 裁剪空间

顶点接下来要从观察空间转换到**裁剪空间**（**clip space**，也被称为**齐次裁剪空间**）中，这个用于变换的矩阵叫做**裁剪矩阵**（**clip matrix**），也被称为**投影矩阵**（**projection matrix**）。

裁剪空间的目标是能够方便地对渲染图元进行裁剪：完全位于这块空间内部的图元将会被保留，完全位于这块空间外部的图元将会被剔除，而与这块空间边界相交的图元就会被裁剪。那么，这块空间是如何决定的呢？答案是由**视锥体（view frustum）**来决定。

视锥体指的是空间中一块区域，这块区域决定了摄像机可以看到的空间。视锥体由六个平面包围而成，这些平面也被称为**裁剪平面（clip planes）**。视锥体有两种类型，这涉及两种投影类型：一种是**正交投影（orthographic projection）**，一种是**透视投影（perspective projection）**。图4.36显示了从同一位置、同一角度渲染同一个场景的两种摄像机的渲染结果。

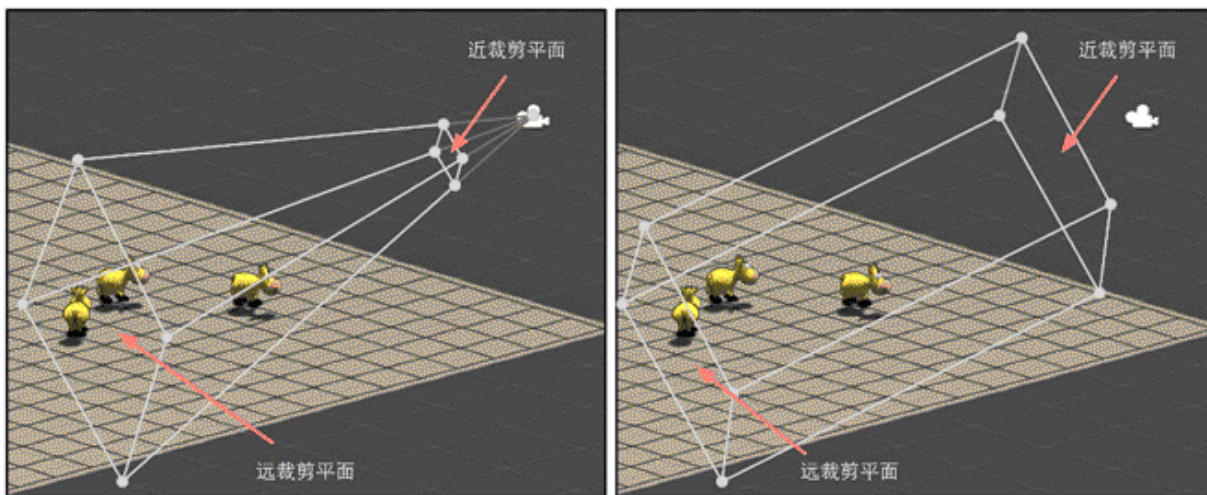


▲图4.36 透视投影（左图）和正交投影（右图）。左下角分别显示了当前摄像机的投影模式和相关属性

从图中可以发现，在透视投影中，地板上的平行线并不会保持平行，离摄像机越近网格越大，离摄像机越远网格越小。而在正交投影中，所有的网格大小都一样，而且平行线会一直保持平行。可以注意到，透视投影模拟了人眼看世界的方式，而正交投影则完全保留了物

体的距离和角度。因此，在追求真实感的3D游戏中我们往往会使用透视投影，而在一些2D游戏或渲染小地图等其他HUD元素时，我们会使用正交投影。

在视锥体的6块裁剪平面中，有两块裁剪平面比较特殊，它们分别被称为**近剪裁平面（near clip plane）**和**远剪裁平面（far clip plane）**。它们决定了摄像机可以看到的深度范围。正交投影和透视投影的视锥体如图4.37所示。



▲图4.37 视锥体和裁剪平面。左图显示了透视投影的视锥体，右图显示了正交投影的视锥体

由图4.37可以看出，透视投影的视锥体是一个金字塔形，侧面的4个裁剪平面将会在摄像机处相交。它更符合视锥体这个词。正交投影的视锥体是一个长方体。前面讲到，我们希望根据视锥体围成的区域对图元进行裁剪，但是，如果直接使用视锥体定义的空间来进行裁剪，那么不同的视锥体就需要不同的处理过程，而且对于透视投影的视锥体来说，想要判断一个顶点是否处于一个金字塔内部是比较麻烦的。因此，我们想用一种更加通用、方便和整洁的方式来进行裁剪的

工作，这种方式就是通过一个投影矩阵把顶点转换到一个裁剪空间中。

投影矩阵有两个目的：

- 首先是为投影做准备。这是个迷惑点，虽然投影矩阵的名称包含了投影二字，但是它并没有进行真正的投影工作，而是在为投影做准备。真正的投影发生在后面的**齐次除法**（**homogeneous division**）过程中。而经过投影矩阵的变换后，顶点的 $w$ 分量将会具有特殊的意义。

读者：投影到底是什么意思呢？

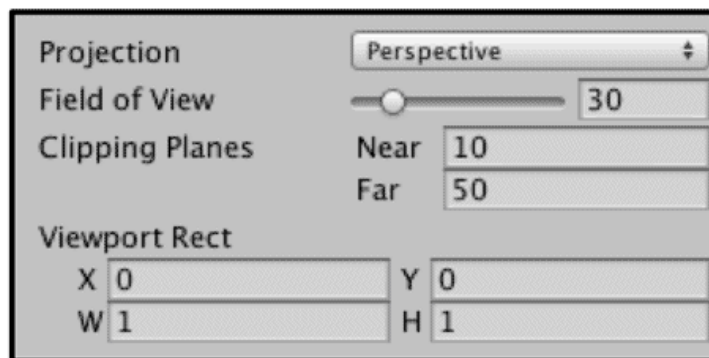
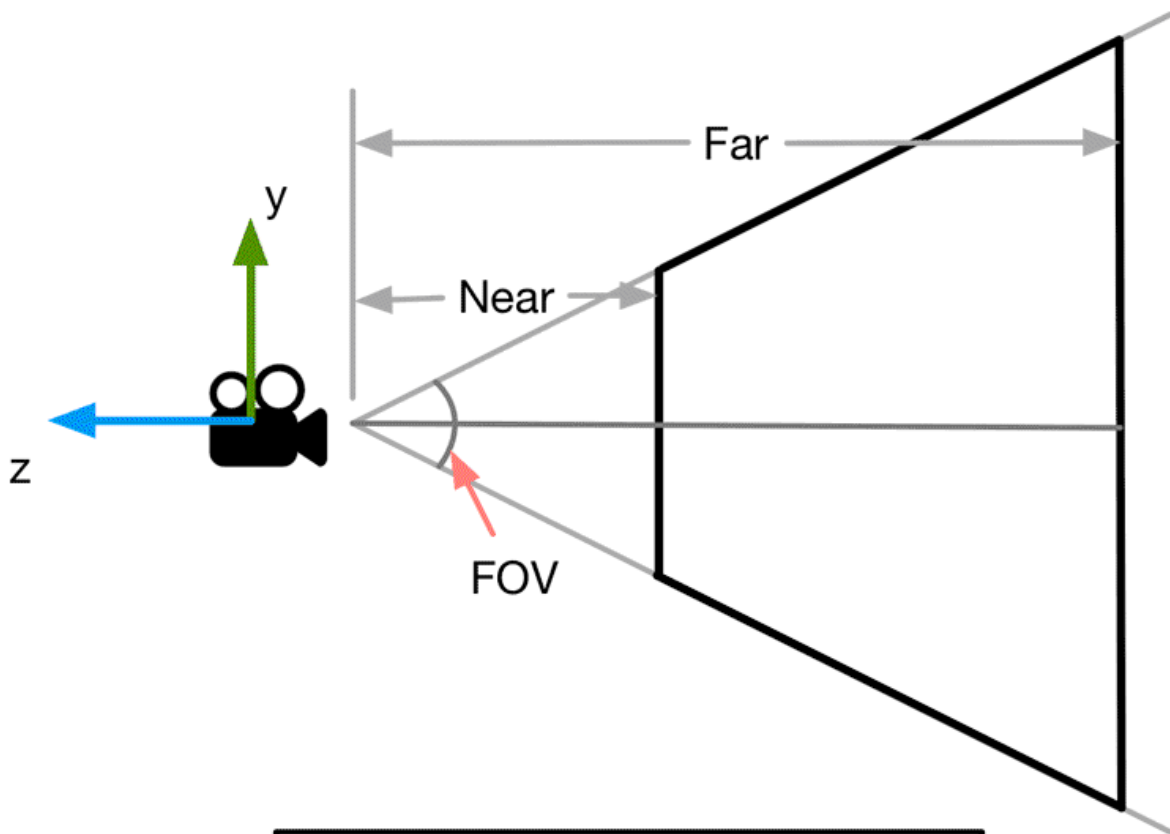
我们：可以理解成是一个空间的降维，例如从四维空间投影到三维空间中。而投影矩阵实际上并不会真的进行这个步骤，它会为真正的投影做准备工作。真正的投影会在屏幕映射时发生，通过齐次除法来得到二维坐标。具体会在4.6.8节中讲到。

- 其次是对 $x$ 、 $y$ 、 $z$ 分量进行缩放。我们上面讲过直接使用视锥体的6个裁剪平面来进行裁剪会比较麻烦。而经过投影矩阵的缩放后，我们可以直接使用 $w$ 分量作为一个范围值，如果 $x$ 、 $y$ 、 $z$ 分量都位于这个范围内，就说明该顶点位于裁剪空间内。

在裁剪空间之前，虽然我们使用了齐次坐标来表示点和矢量，但它们的第四个分量都是固定的：点的 $w$ 分量是1，方向矢量的 $w$ 分量是0。经过投影矩阵的变换后，我们会赋予齐次坐标的第4个坐标更加丰富的含义。下面，我们来看一下两种投影类型使用的投影矩阵具体是什么。

## 1. 透视投影

视锥体的意义在于定义了场景中的一块三维空间。所有位于这块空间内的物体将会被渲染，否则就会被剔除或裁剪。我们已经知道，这块区域由6个裁剪平面定义，那么这6个裁剪平面又是怎么决定的呢？在Unity中，它们由Camera组件中的参数和Game视图的纵横比共同决定，如图4.38所示。



▲图4.38 透视摄像机的参数对透视投影视锥体的影响

由图4.38可以看出，我们可以通过Camera组件的Field of View（简称FOV）属性来改变视锥体竖直方向的张开角度，而Clipping Planes中的Near和Far参数可以控制视锥体的近裁剪平面和远裁剪平面距离摄像机的远近。这样，我们可以求出视锥体近裁剪平面和远裁剪平面的高度，也就是：

$$nearClipPlaneHeight = 2 \cdot Near \cdot \tan \frac{FOV}{2}$$

现在我们还缺乏横向的信息。这可以通过摄像机的横纵比得到。在Unity中，一个摄像机的横纵比由Game视图的横纵比和Viewport Rect中的W和H属性共同决定（实际上，Unity允许我们在脚本里通过Camera.aspect进行更改，但这里不做讨论）。假设，当前摄像机的横纵比为Aspect，我们定义：

$$Aspect = \frac{nearClipPlaneWidth}{nearClipPlaneHeight}$$

$$Aspect = \frac{farClipPlaneWidth}{farClipPlaneHeight}$$

现在，我们可以根据已知的Near、Far、FOV和Aspect的值来确定透视投影的投影矩阵。如下：

$$M_{frustum} = \begin{bmatrix} \frac{\cot \frac{FOV}{2}}{Aspect} & 0 & 0 & 0 \\ 0 & \cot \frac{FOV}{2} & 0 & 0 \\ 0 & 0 & -\frac{Far+Near}{Far-Near} & -\frac{2 \cdot Near \cdot Far}{Far-Near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

上面公式的推导部分可以参见本章的扩展阅读部分。需要注意的是，这里的投影矩阵是建立在Unity对坐标系的假定上面的，也就是说，我们针对的是观察空间为右手坐标系，使用列矩阵在矩阵右侧进行相乘，且变换后z分量范围将在 $[-w, w]$ 之间的情况。而在类似DirectX这样的图形接口中，它们希望变换后z分量范围将在 $[0, w]$ 之间，因此就需要对上面的透视矩阵进行一些更改。这不在本书的讨论范围内。

而一个顶点和上述投影矩阵相乘后，可以由观察空间变换到裁剪空间中，结果如下：

$$\begin{aligned} \mathbf{P}_{clip} &= \mathbf{M}_{frustum} \mathbf{P}_{view} \\ &= \begin{bmatrix} \frac{\cot \frac{FoV}{2}}{Aspect} & 0 & 0 & 0 \\ 0 & \cot \frac{FoV}{2} & 0 & 0 \\ 0 & 0 & -\frac{Far+Near}{Far-Near} & -\frac{2 \cdot Near \cdot Far}{Far-Near} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} x \frac{\cot \frac{FoV}{2}}{Aspect} \\ y \cot \frac{FoV}{2} \\ -z \frac{Far+Near}{Far-Near} - \frac{2 \cdot Near \cdot Far}{Far-Near} \\ -z \end{bmatrix} \end{aligned}$$

从结果可以看出，这个投影矩阵本质就是对x、y和z分量进行了不同程度的缩放（当然，z分量还做了一个平移），缩放的目的是为了便于裁剪。我们可以注意到，此时顶点的w分量不再是1，而是原先z分量



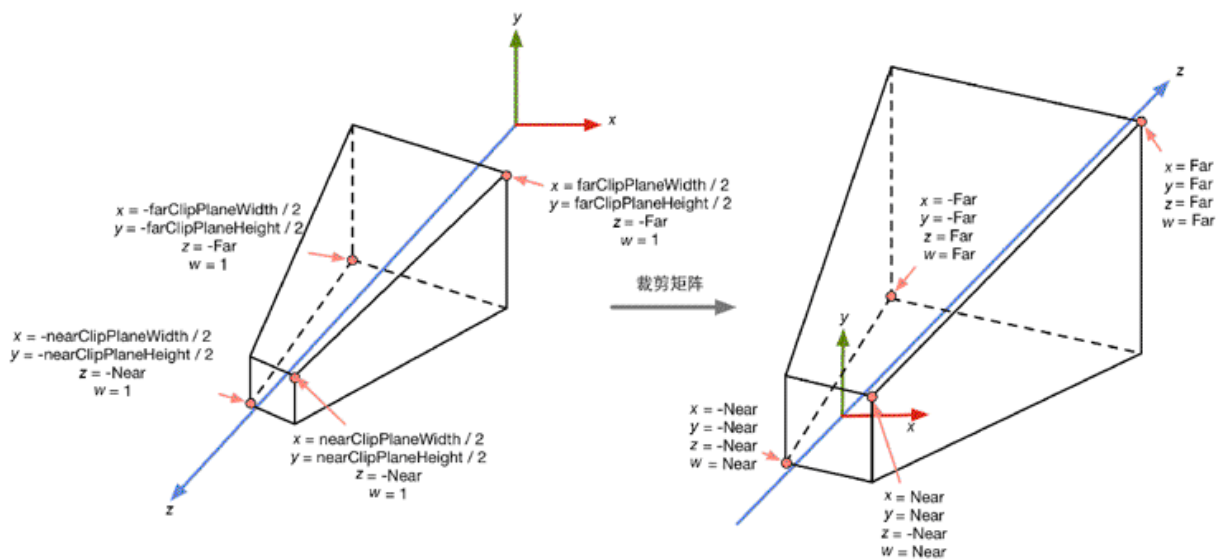
的取反结果。现在,我们就可以按如下不等式来判断一个变换后的顶点是否位于视锥体内。如果一个顶点在视锥体内, 那么它变换后的坐标必须满足:

$$-w \leq x \leq w$$

$$-w \leq y \leq w$$

$$-w \leq z \leq w$$

任何不满足上述条件的图元都需要被剔除或者裁剪。图4.39显示了经过上述投影矩阵后, 视锥体的变化。

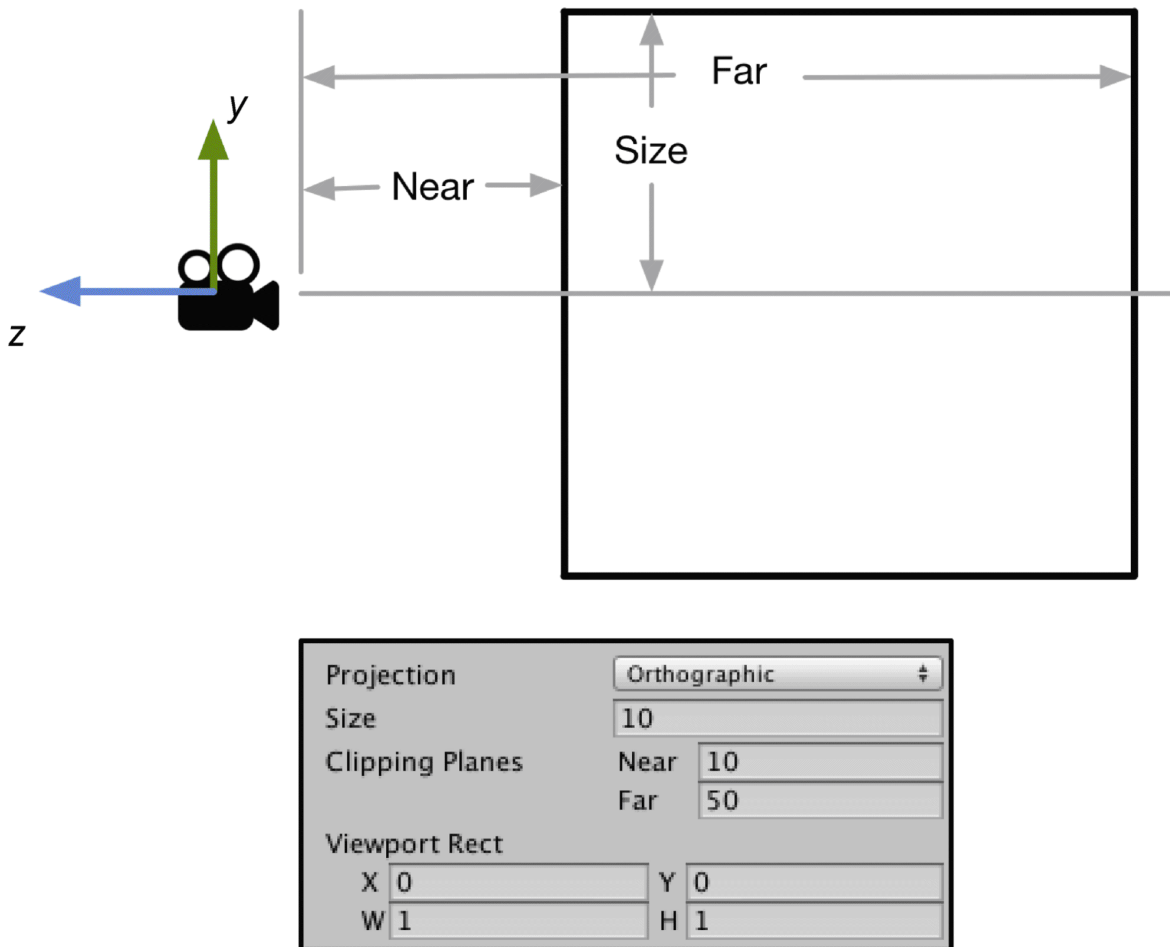


▲ 图4.39 在透视投影中, 投影矩阵对顶点进行了缩放。图中标注了4个关键点经过投影矩阵变换后的结果。从这些结果可以看出x、y、z和w分量的范围发生的变化

从图4.39还可以注意到, 裁剪矩阵会改变空间的旋向性: 空间从右手坐标系变换到了左手坐标系。这意味着, 离摄像机越远, z值将越大。

## 2. 正交投影

首先，我们还是看一下正交投影中的6个裁剪平面是如何定义的。和透视投影类似，在Unity中，它们也是由Camera组件中的参数和Game视图的纵横比共同决定，如图4.40所示。



▲图4.40 正交摄像机的参数对正交投影视锥体的影响

正交投影的视锥体是一个长方体，因此计算上相比透视投影来说更加简单。由图可以看出，我们可以通过Camera组件的Size属性来改变视锥体竖直方向上高度的一半，而Clipping Planes中的Near和Far参数可

以控制视锥体的近裁剪平面和远裁剪平面距离摄像机的远近。这样，我们可以求出视锥体近裁剪平面和远裁剪平面的高度，也就是：

$$nearClipPlaneHeight = 2 \cdot Size$$

$$farClipPlaneHeight = nearClipPlaneHeight$$

现在我们还缺乏横向的信息。同样，我们可以通过摄像机的纵横比得到。假设，当前摄像机的纵横比为Aspect，那么：

$$nearClipPlaneWidth = Aspect \cdot nearClipPlaneHeight$$

$$farClipPlaneWidth = nearClipPlaneWidth$$

现在，我们可以根据已知的Near、Far、Size和Aspect的值来确定正交投影的裁剪矩阵。如下：

$$\mathbf{M}_{ortho} = \begin{bmatrix} \frac{1}{Aspect \cdot Size} & 0 & 0 & 0 \\ 0 & \frac{1}{Size} & 0 & 0 \\ 0 & 0 & -\frac{2}{Far - Near} & -\frac{Far + Near}{Far - Near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

上面公式的推导部分可以参见本章的扩展阅读部分。同样，这里的投影矩阵是建立在Unity对坐标系的假定上面的。

一个顶点和上述投影矩阵相乘后的结果如下：

$$\mathbf{P}_{clip} = \mathbf{M}_{ortho} \mathbf{P}_{view}$$

$$\begin{aligned}
&= \begin{bmatrix} \frac{1}{Aspect \cdot Size} & 0 & 0 & 0 \\ 0 & \frac{1}{Size} & 0 & 0 \\ 0 & 0 & -\frac{2}{Far-Near} & -\frac{Far+Near}{Far-Near} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} \frac{x}{Aspect \cdot Size} \\ \frac{y}{Size} \\ -\frac{2z}{Far-Near} - \frac{Far+Near}{Far-Near} \\ 1 \end{bmatrix}
\end{aligned}$$

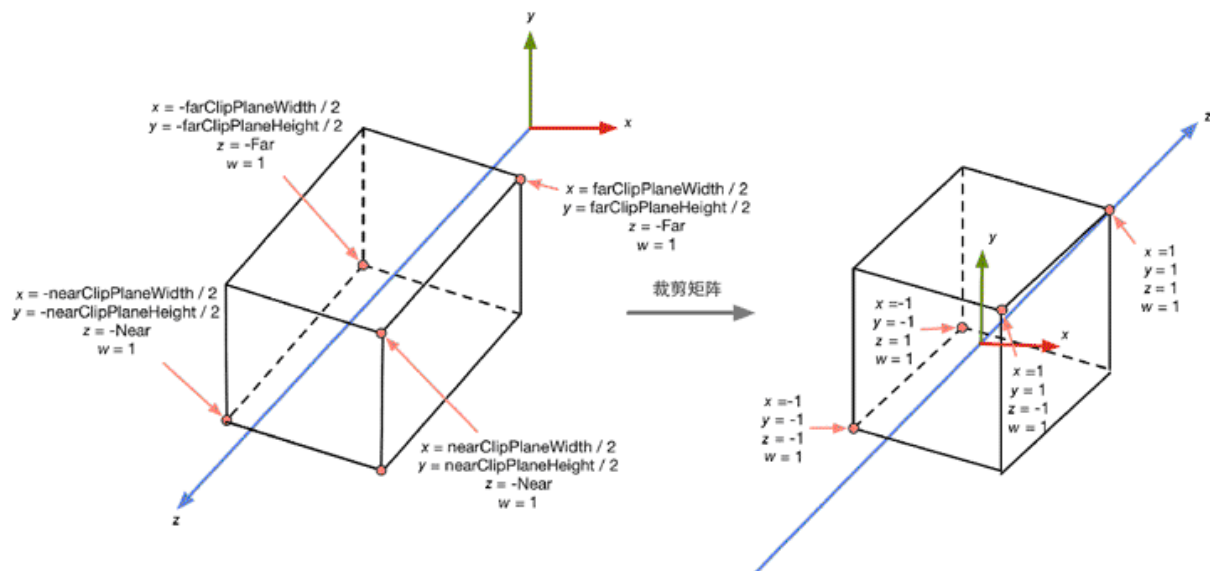
注意到，和透视投影不同的是，使用正交投影的投影矩阵对顶点进行变换后，其w分量仍然为1。本质是因为投影矩阵最后一行的不同，透视投影的投影矩阵的最后一行是 $[0 \ 0 \ -1 \ 0]$ ，而正交投影的投影矩阵的最后一行是 $[0 \ 0 \ 0 \ 1]$ 。这样的选择是有原因的，是为了为齐次除法做准备。具体会在下一节中讲到。

判断一个变换后的顶点是否位于视锥体内使用的不等式和透视投影中的一样，这种通用性也是为什么要使用投影矩阵的原因之一。图4.41显示了经过上述投影矩阵后，正交投影的视锥体的变化。

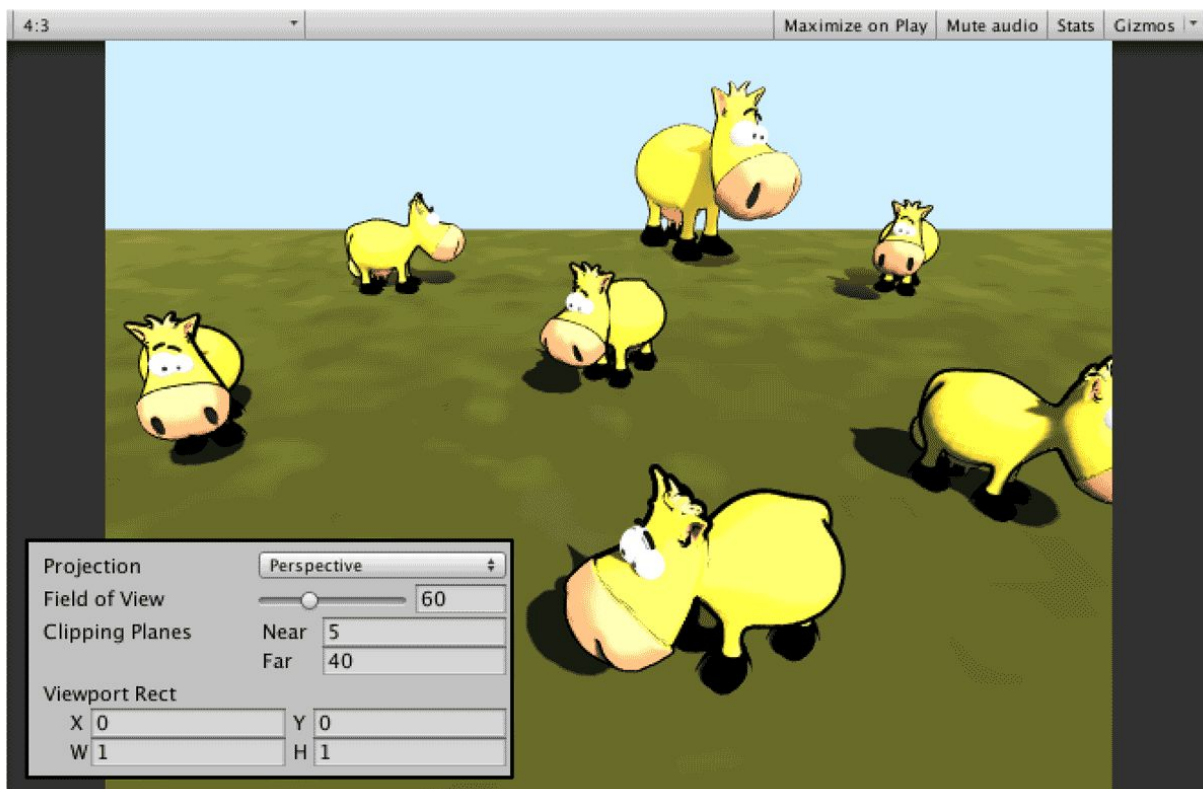
同样，裁剪矩阵改变了空间的旋向性。可以注意到，经过正交投影变换后的顶点实际已经位于一个立方体内了。

希望看到这里读者的脑袋还没有爆炸。现在，我们继续来看我们的农场游戏。在4.6.6节的最后，我们已经帮助妞妞确定了它的鼻子在观察空间中的位置—— $(9, 8.84, -27.31)$ 。现在，我们要计算它在裁剪空间中的位置。

首先，我们需要知道农场游戏中使用的摄像机类型。由于农场游戏是一个3D游戏，因此这里我们使用了透视摄像机。摄像机参数和Game视图的横纵比如图4.42所示。



▲ 图4.41 在正交投影中，投影矩阵对顶点进行了缩放。图中标注了4个关键点经过投影矩阵变换后的结果。从这些结果可以看出x、y、z和w分量范围发生的变化。



▲ 图4.42 农场游戏使用的摄像机参数和游戏画面的横纵比

据此，我们可以知道透视投影的参数：FOV为60°，Near为5，Far为40，Aspect为4/3 = 1.333。那么，对应的投影矩阵就是：

$$M_{frustum} = \begin{bmatrix} \frac{\cot \frac{FOV}{2}}{Aspect} & 0 & 0 & 0 \\ 0 & \cot \frac{FOV}{2} & 0 & 0 \\ 0 & 0 & -\frac{Far+Near}{Far-Near} & -\frac{2 \cdot Near \cdot Far}{Far-Near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 1.299 & 0 & 0 & 0 \\ 0 & 1.732 & 0 & 0 \\ 0 & 0 & -1.286 & -11.429 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

然后，我们用这个投影矩阵来把妞妞的鼻子从观察空间转换到裁剪空间中。如下：

$$\mathbf{P}_{clip} = \mathbf{M}_{frustum} \mathbf{P}_{view}$$

$$= \begin{bmatrix} 1.299 & 0 & 0 & 0 \\ 0 & 1.732 & 0 & 0 \\ 0 & 0 & -1.286 & -11.429 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 9 \\ 8.84 \\ -27.31 \\ 1 \end{bmatrix} = \begin{bmatrix} 11.691 \\ 15.311 \\ 23.692 \\ 27.31 \end{bmatrix}$$

这样，我们就求出了妞妞的鼻子在裁剪空间中的位置——(11.691, 15.311, 23.692, 27.31)。接下来，Unity会判断妞妞的鼻子是否需要裁剪。通过比较得到，妞妞的鼻子满足下面的不等式：

$$-w \leq x \leq w \rightarrow -27.31 \leq 11.691 \leq 27.31$$

$$-w \leq y \leq w \rightarrow -27.31 \leq 15.311 \leq 27.31$$

$$-w \leq z \leq w \rightarrow -27.31 \leq 23.692 \leq 27.31$$

由此，我们可以判断，妞妞的鼻子位于视锥体内，不需要被裁剪。

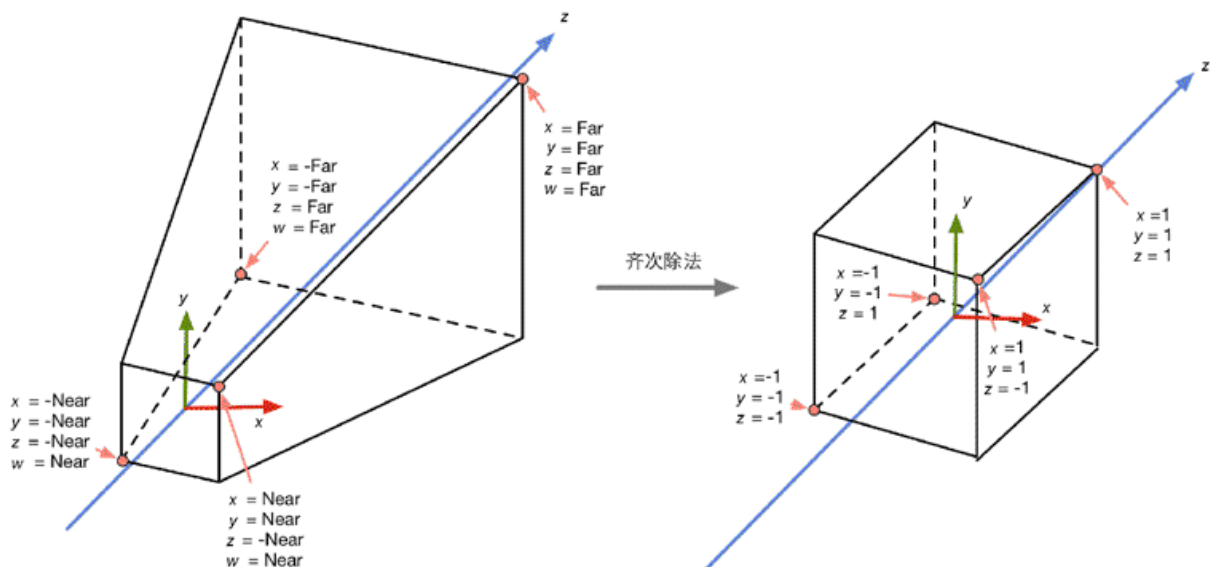
#### 4.6.8 屏幕空间

经过投影矩阵的变换后，我们可以进行裁剪操作。当完成了所有的裁剪工作后，就需要进行真正的投影了，也就是说，我们需要把视锥体投影到**屏幕空间（screen space）**中。经过这一步变换，我们会得到真正的像素位置，而不是虚拟的三维坐标。

屏幕空间是一个二维空间，因此，我们必须把顶点从裁剪空间投影到屏幕空间中，来生成对应的2D坐标。这个过程可以理解成有两个步骤。

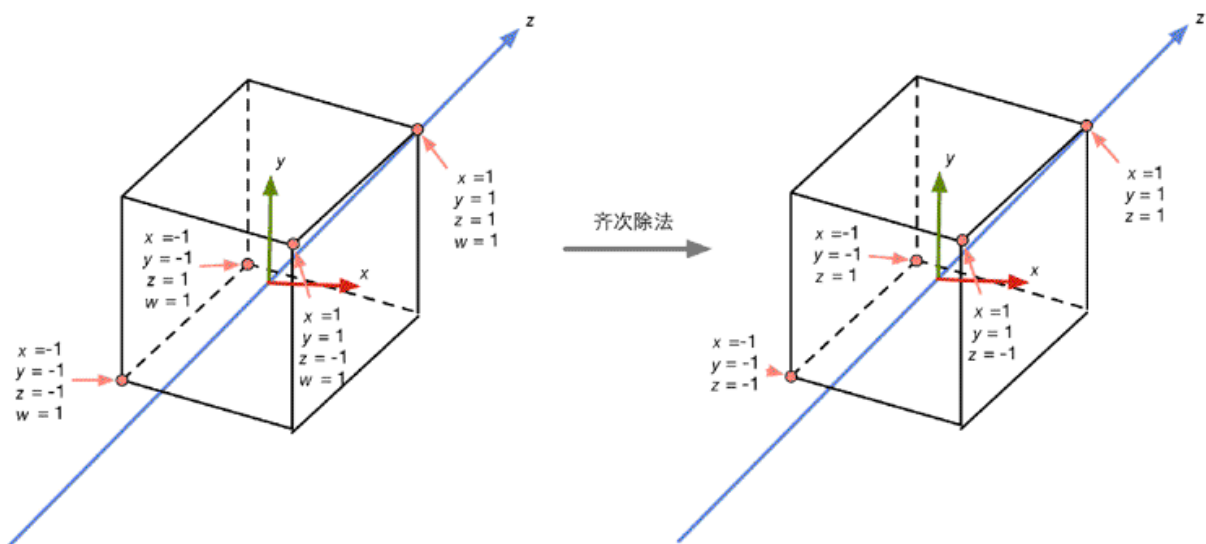
首先，我们需要进行标准**齐次除法（homogeneous division）**，也被称为**透视除法（perspective division）**。虽然这个步骤听起来很陌生，但是它实际上非常简单，就是用齐次坐标系的 $w$ 分量去除 $x$ 、 $y$ 、 $z$ 分量。在OpenGL中，我们把这一步得到的坐标叫做**归一化的设备坐标（Normalized Device Coordinates, NDC）**。经过这一步，我们可以把坐标从齐次裁剪坐标空间转换到NDC中。经过透视投影变换后的裁剪空间，经过齐次除法后会变换到一个立方体内。按照OpenGL的传统，这个立方体的 $x$ 、 $y$ 、 $z$ 分量的范围都是 $[-1, 1]$ 。但在DirectX这样的API中， $z$ 分量的范围会是 $[0, 1]$ 。而Unity选择了OpenGL这样的齐次裁剪空间。如图4.43所示。





▲ 图4.43 经过齐次除法后，透视投影的裁剪空间会变换到一个立方体

而对于正交投影来说，它的裁剪空间实际已经是一个立方体了，而且由于经过正交投影矩阵变换后的顶点的 $w$ 分量是1，因此齐次除法并不会对顶点的 $x$ 、 $y$ 、 $z$ 坐标产生影响。如图4.44所示。



▲ 图4.44 经过齐次除法后，正交投影的裁剪空间会变换到一个立方体

经过齐次除法后，透视投影和正交投影的视锥体都变换到一个相同的立方体内。现在，我们可以根据变换后的 $x$ 和 $y$ 坐标来映射输出窗口的对应像素坐标。

在Unity中，屏幕空间左下角的像素坐标是 $(0, 0)$ ，右上角的像素坐标是 $(\text{pixelWidth}, \text{pixelHeight})$ 。由于当前 $x$ 和 $y$ 坐标都是 $[-1, 1]$ ，因此这个映射的过程就是一个缩放的过程。

齐次除法和屏幕映射的过程可以使用下面的公式来总结：

上面的式子对 $x$ 和 $y$ 分量都进行了处理，那么 $z$ 分量呢？通常， $z$ 分量会被用于深度缓冲。一个传统的方式是把 $\frac{\text{clip}_z}{\text{clip}_w}$ 的值直接存进深度缓冲中，但这并不是必须的。通常驱动生产商会根据硬件来选择最好的存储格式。此时 $\text{clip}_w$ 也并不会被抛弃，虽然它已经完成了它的主要工作——在齐次除法中作为分母来得到NDC，但它仍然会在后续的一些工作中起到重要的作用，例如进行透视校正插值。

在Unity中，从裁剪空间到屏幕空间的转换是由底层帮我们完成的。我们的顶点着色器只需要把顶点转换到裁剪空间即可。

在上一步中，我们知道了裁剪空间中妞妞鼻子的位置—— $(11.691, 15.311, 23.692, 27.31)$ 。现在，我们终于可以确定妞妞的鼻子在屏幕上的像素位置。假设，当前屏幕的像素宽度为400，高度为300。首先，

我们需要进行齐次除法，把裁剪空间的坐标投影到NDC中。然后，再映射到屏幕空间中。这个过程如下：

$$\begin{aligned} screen_x &= \frac{clip_x \cdot pixelWidth}{2 \cdot clip_w} + \frac{pixelWidth}{2} \\ &= \frac{11.691 \cdot 400}{2 \cdot 27.31} + \frac{400}{2} \\ &= 285.617 \\ screen_y &= \frac{clip_y \cdot pixelHeight}{2 \cdot clip_h} + \frac{pixelHeight}{2} \\ &= \frac{15.311 \cdot 300}{2 \cdot 27.31} + \frac{300}{2} \\ &= 234.096 \end{aligned}$$

由此，我们知道了妞妞鼻子在屏幕上的位置——(285.617, 234.096)。

#### 4.6.9 总结

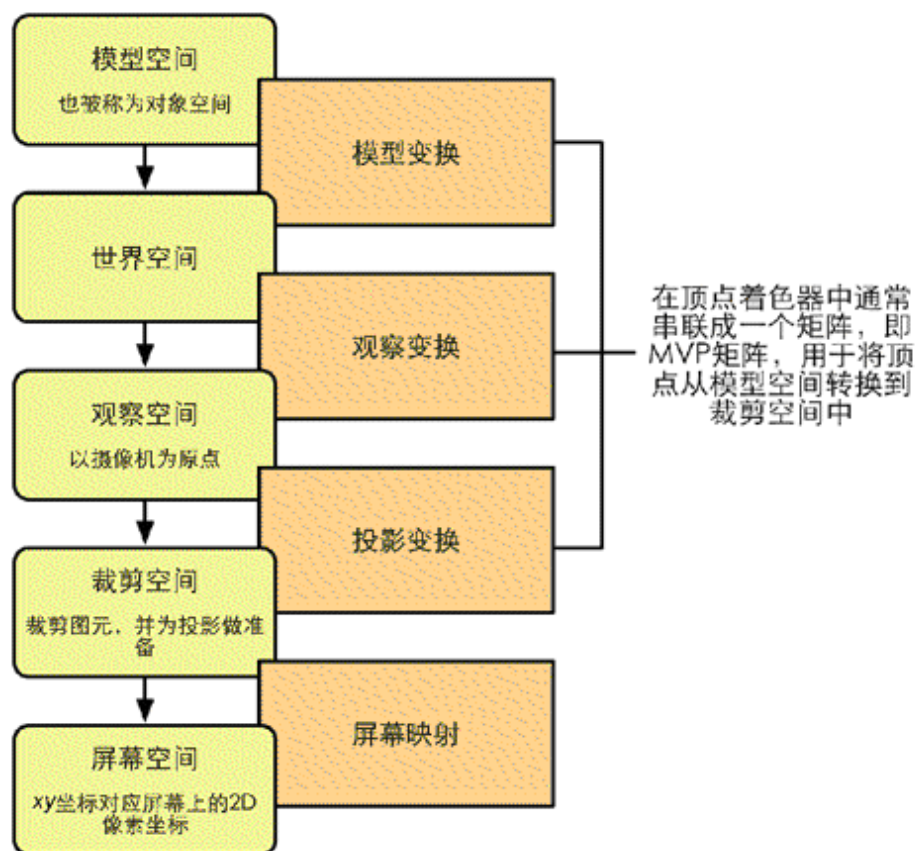
以上就是一个顶点如何从模型空间变换到屏幕坐标的过程。图4.45总结了这些空间和用于变换的矩阵。

顶点着色器的最基本的任务就是把顶点坐标从模型空间转换到裁剪空间中。这对应了图4.45中的前3个顶点变换过程。而在片元着色器中，我们通常也可以得到该片元在屏幕空间的像素位置。我们会在4.9.3节中看到如何得到这些像素位置。

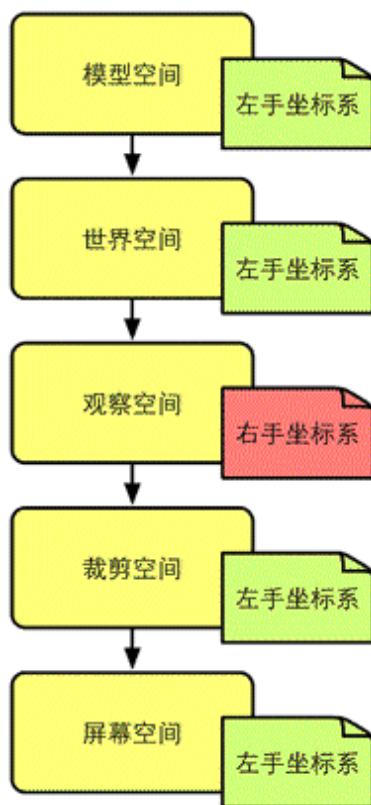
在Unity中，坐标系的旋向性也随着变换发生了改变。图4.46总结了Unity中各个空间使用的坐标系旋向性。

从图4.46中可以发现，只有在观察空间中Unity使用了右手坐标系。

需要注意的是，这里仅仅给出的是一些最重要的坐标空间。还有一些空间在实际开发中也会遇到，例如**切线空间（tangent space）**。切线空间通常用于法线映射，在后面的4.7节中我们会讲到。



▲ 图4.45 渲染流水线中顶点的空间变换过程



▲图4.46 Unity中各个坐标空间的旋向性

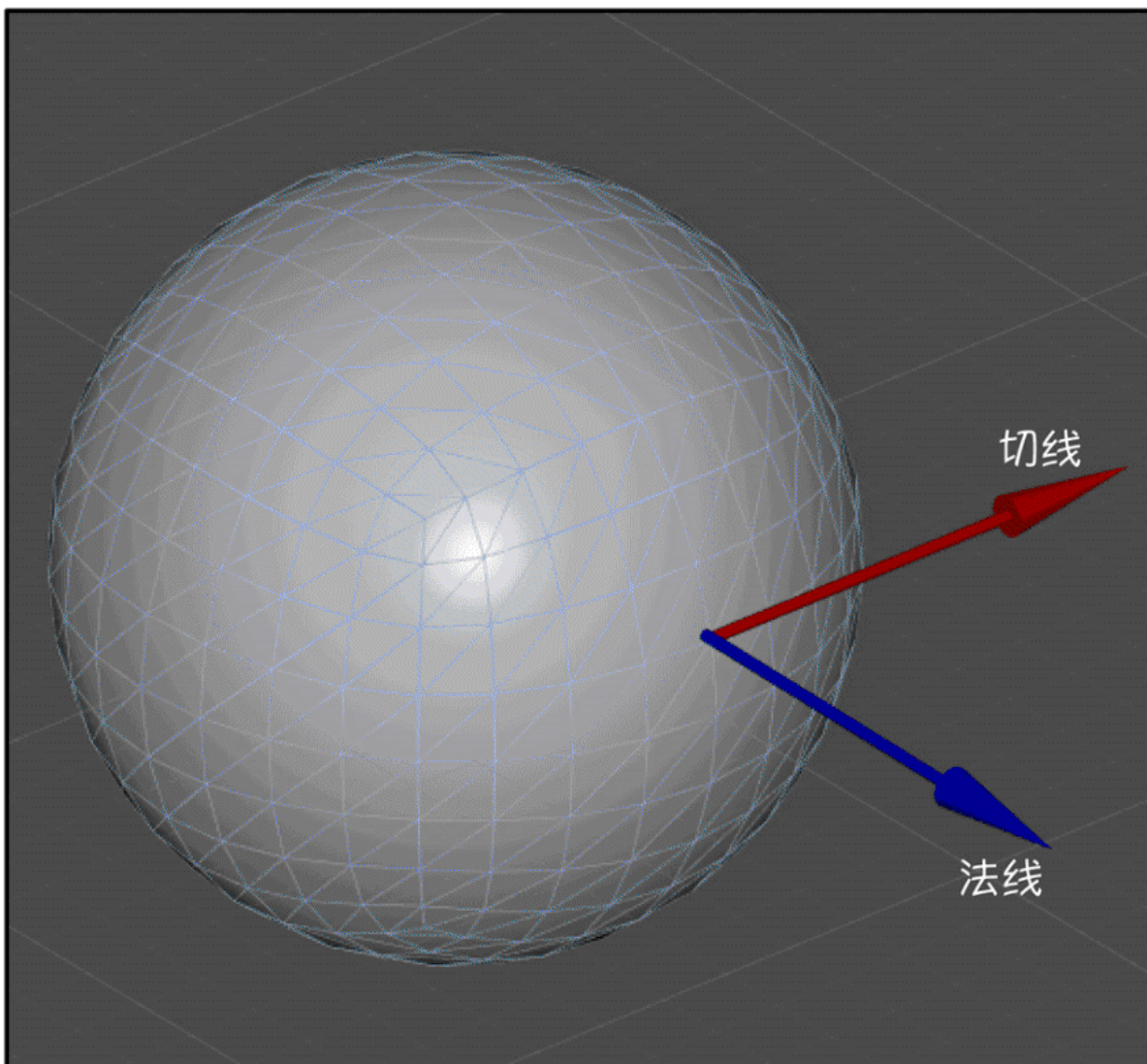
## 4.7 法线变换

在本章的最后，我们来看一种特殊的变换：法线变换。

**法线（normal）**，也被称为**法向量（normal vector）**。在上面我们已经看到如何使用变换矩阵来变换一个顶点或一个方向矢量，但法线是需要我们特殊处理的一种方向矢量。在游戏中，模型的一个顶点往往会携带额外的信息，而顶点法线就是其中一种信息。当我们变换一个模型的时候，不仅需要变换它的顶点，还需要变换顶点法线，以便在后续处理（如片元着色器）中计算光照等。

一般来说，点和绝大部分方向矢量都可以使用同一个 $4 \times 4$ 或 $3 \times 3$ 的变换矩阵 $\mathbf{M}_{A \rightarrow B}$ 将其从坐标空间 $\mathbf{A}$ 变换到坐标空间 $\mathbf{B}$ 中。但在变换法线的时候，如果使用同一个变换矩阵，可能就无法确保维持法线的垂直性。下面就来了解一下为什么会出现这样的问题。

我们先来了解一下另一种方向矢量——**切线（tangent）**，也被称为**切向量（tangent vector）**。与法线类似，切线往往也是模型顶点携带的一种信息。它通常与纹理空间对齐，而且与法线方向垂直，如图4.47所示。



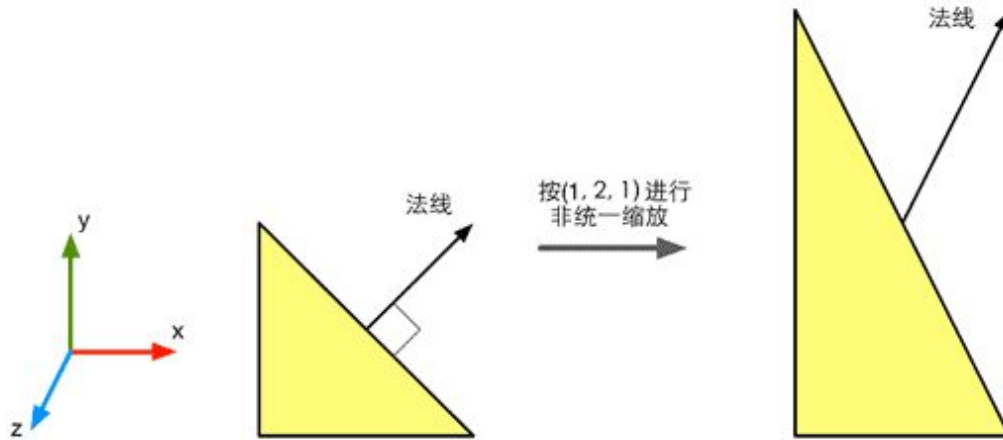
▲图4.47 顶点的切线和法线。切线和法线互相垂直

由于切线是由两个顶点之间的差值计算得到的，因此我们可以直接使用用于变换顶点的变换矩阵来变换切线。假设，我们使用 $3 \times 3$ 的变换矩阵 $\mathbf{M}_{A \rightarrow B}$ 来变换顶点（注意，这里涉及的变换矩阵都是 $3 \times 3$ 的矩阵，不考虑平移变换。这是因为切线和法线都是方向矢量，不会受平移的影响），可以由下面的式子直接得到变换后的切线：



$$\mathbf{T}_B = \mathbf{M}_{A \rightarrow B} \mathbf{T}_A$$

其中 $\mathbf{T}_A$ 和 $\mathbf{T}_B$ 分别表示在坐标空间A下和坐标空间B下的切线方向。但如果直接使用 $\mathbf{M}_{A \rightarrow B}$ 来变换法线，得到的新的法线方向可能就不会与表面垂直了。图4.48给出了这样的例子。



▲图4.48 进行非统一缩放时，如果使用和变换顶点相同的变换矩阵来变换法线，就会得到错误的结果，即变换后的法线方向与平面不再垂直

那么，应该使用哪个矩阵来变换法线呢？我们可以由数学约束条件来推出这个矩阵。我们知道同一个顶点的切线 $\mathbf{T}_A$ 和法线 $\mathbf{N}_A$ 必须满足垂直条件，即 $\mathbf{T}_A \cdot \mathbf{N}_A = 0$ 。给定变换矩阵 $\mathbf{M}_{A \rightarrow B}$ ，我们已经知道 $\mathbf{T}_B = \mathbf{M}_{A \rightarrow B} \mathbf{T}_A$ 。我们现在想要找到一个矩阵 $\mathbf{G}$ 来变换法线 $\mathbf{N}_A$ ，使得变换后的法线仍然与切线垂直。即

$$\mathbf{T}_B \cdot \mathbf{N}_B = (\mathbf{M}_{A \rightarrow B} \mathbf{T}_A) \cdot (\mathbf{G} \mathbf{N}_A) = 0$$

对上式进行一些推导后可得

$$(\mathbf{M}_{A \rightarrow B} \mathbf{T}_A) \cdot (\mathbf{G} \mathbf{N}_A) = (\mathbf{M}_{A \rightarrow B} \mathbf{T}_A)^T (\mathbf{G} \mathbf{N}_A) = \mathbf{T}_A^T \mathbf{M}_{A \rightarrow B}^T \mathbf{G} \mathbf{N}_A = \mathbf{T}_A^T (\mathbf{M}_{A \rightarrow B}^T \mathbf{G}) \mathbf{N}_A =$$

0

由于 $\mathbf{T}_A \cdot \mathbf{N}_A = 0$ ，因此如果 $\mathbf{M}_{A \rightarrow B}^T \mathbf{G} = \mathbf{I}$ ，那么上式即可成立。也就是说，如果 $\mathbf{G} = (\mathbf{M}_{A \rightarrow B}^T)^{-1} = (\mathbf{M}_{A \rightarrow B}^{-1})^T$ ，即使用原变换矩阵的逆转置矩阵来变换法线就可以得到正确的结果。

值得注意的是，如果变换矩阵 $\mathbf{M}_{A \rightarrow B}$ 是正交矩阵，那么 $\mathbf{M}_{A \rightarrow B}^{-1} = \mathbf{M}_{A \rightarrow B}^T$ ，因此 $(\mathbf{M}_{A \rightarrow B}^T)^{-1} = \mathbf{M}_{A \rightarrow B}$ ，也就是说我们可以使用用于变换顶点的变换矩阵来直接变换法线。如果变换只包括旋转变换，那么这个变换矩阵就是正交矩阵。而如果变换只包含旋转和统一缩放，而不包含非统一缩放，我们利用统一缩放系数 $k$ 来得到变换矩阵 $\mathbf{M}_{A \rightarrow B}$ 的逆转置矩阵 $(\mathbf{M}_{A \rightarrow B}^T)^{-1} = \frac{1}{k} \mathbf{M}_{A \rightarrow B}$ 。这样就可以避免计算逆矩阵的过程。如果变换中包含了非统一变换，那么我们就必须要求解逆矩阵来得到变换法线的矩阵。

## 4.8 Unity Shader的内置变量（数学篇）

使用Unity写Shader的一个好处在于，它提供了很多内置的参数，这使得我们不再需要自己手动计算一些值。本节将给出Unity内置的用于空间变换和摄像机以及屏幕参数的内置变量。这些内置变量可以在UnityShaderVariables.cginc文件中找到定义和说明。

### 4.8.1 变换矩阵

首先是用于坐标空间变换的矩阵。表4.2给出了Unity 5.2版本提供的所有内置变换矩阵。下面所有的矩阵都是float4x4类型的。

读者：为什么在我的Unity中，有些变量不存在呢？

我们：可能是由于你使用的Unity版本和本书使用的版本不同。在写本书时，我们使用的Unity版本是最新的5.2。而在4.x版本中，一些内置变量可能会与之不同。

**表4.2     Unity内置的变换矩阵**

变量名	描述
UNITY_MATRIX_MVP	当前的模型观察投影矩阵，用于将顶点/方向矢量从模型空间变换到裁剪空间
UNITY_MATRIX_MV	当前的模型观察矩阵，用于将顶点/方向矢量从模型空间变换到观察空间
UNITY_MATRIX_V	当前的观察矩阵，用于将顶点/方向矢量从世界空间变换到观察空间
UNITY_MATRIX_P	当前的投影矩阵，用于将顶点/方向矢量从观察空间变换到裁剪空间
UNITY_MATRIX_VP	当前的观察投影矩阵，用于将顶点/方向矢量从世界空间变换到裁剪空间
UNITY_MATRIX_T_MV	UNITY_MATRIX_MV的转置矩阵
UNITY_MATRIX_IT_MV	UNITY_MATRIX_MV的逆转置矩阵，用于将法线从模型空间变换到观察空间，也可用于得到UNITY_MATRIX_MV的逆矩阵

变量名	描述
<code>_Object2World</code>	当前的模型矩阵，用于将顶点/方向矢量从模型空间变换到世界空间
<code>_World2Object</code>	<code>_Object2World</code> 的逆矩阵，用于将顶点/方向矢量从世界空间变换到模型空间

表4.2给出了这些矩阵的常用用法。但读者可以根据需求来达到不同的目的，例如我们可以提取坐标空间的坐标轴，方法可回顾4.6.2节。

其中有一个矩阵比较特殊，即`UNITY_MATRIX_T_MV`矩阵。很多对数学不了解的读者不理解这个矩阵有什么用处。如果读者认真看过矩阵一节的知识，应该还会记得一种非常吸引人的矩阵类型——正交矩阵。对于正交矩阵来说，它的逆矩阵就是转置矩阵。因此，如果`UNITY_MATRIX_MV`是一个正交矩阵的话，那么`UNITY_MATRIX_T_MV`就是它的逆矩阵，也就是说，我们可以使用`UNITY_MATRIX_T_MV`把顶点和方向矢量从观察空间变换到模型空间。那么问题是，`UNITY_MATRIX_MV`什么时候是一个正交矩阵呢？读者可以从4.5节找到答案。总结一下，如果我们只考虑旋转、平移和缩放这3种变换的话，如果一个模型的变换只包括旋转，那么`UNITY_MATRIX_MV`就是一个正交矩阵。这个条件似乎有些苛刻，我们可以把条件再放宽一些，如果只包括旋转和统一缩放（假设缩放系数是 $k$ ），那么`UNITY_MATRIX_MV`就几乎是一个正交矩阵了。为什么是几乎呢？因为统一缩放可能会导致每一行（或每一列）的矢量长

度不为1，而是 $k$ ，这不符合正交矩阵的特性，但我们可以通过除以这个统一缩放系数，来把它变成正交矩阵。在这种情况下，

$\text{UNITY\_MATRIX\_MV}$ 的逆矩阵就是 $\frac{1}{k}\text{UNITY\_MATRIX\_T\_MV}$ 。而且，如果我们只是对方向矢量进行变换的话，条件可以放得更宽，即不用考虑有没有平移变换，因为平移对方向矢量没有影响。因此，我们可以截取 $\text{UNITY\_MATRIX\_T\_MV}$ 的前3行前3列来把方向矢量从观察空间变换到模型空间（前提是只存在旋转变换和统一缩放）。对于方向矢量，我们可以在使用前对它们进行归一化处理，来消除统一缩放的影响。

还有一个矩阵需要说明一下，那就是 $\text{UNITY\_MATRIX\_IT\_MV}$ 矩阵。我们在4.7节已经知道，法线的变换需要使用原变换矩阵的逆转置矩阵。因此 $\text{UNITY\_MATRIX\_IT\_MV}$ 可以把法线从模型空间变换到观察空间。但只要我们做一点手脚，它也可以用于直接得到 $\text{UNITY\_MATRIX\_MV}$ 的逆矩阵——我们只需要对它进行转置就可以了。因此，为了把顶点或方向矢量从观察空间变换到模型空间，我们可以使用类似下面的代码：

```
// 方法一：使用transpose函数对UNITY_MATRIX_IT_MV进行转置，
// 得到UNITY_MATRIX_MV的逆矩阵，然后进行列矩阵乘法，
// 把观察空间中的点或方向矢量变换到模型空间中
float4 modelPos = mul(transpose(UNITY_MATRIX_IT_MV), viewPos);

// 方法二：不直接使用转置函数transpose，而是交换mul参数的位置，使用行矩阵乘法
// 本质和方法一是完全一样的
float4 modelPos = mul(viewPos, UNITY_MATRIX_IT_MV);
```

关于mul函数参数位置导致的不同，在4.9.2节中我们会继续讲到。

## 4.8.2 摄像机和屏幕参数

Unity提供了一些内置变量来让我们访问当前正在渲染的摄像机的参数信息。这些参数对应了摄像机上的Camera组件中的属性值。表4.3给出了Unity 5.2版本提供的这些变量。

表4.3 Unity内置的摄像机和屏幕参数

变量名	类型	描述
<code>_WorldSpaceCameraPos</code>	float3	该摄像机在世界空间中的位置
<code>_ProjectionParams</code>	float4	$x = 1.0$ （或 $-1.0$ ，如果正在使用一个翻转的投影矩阵进行渲染）， $y = \text{Near}$ ， $z = \text{Far}$ ， $w = 1.0 + 1.0/\text{Far}$ ，其中Near和Far分别是近裁剪平面和远裁剪平面和摄像机的距离
<code>_ScreenParams</code>	float4	$x = \text{width}$ ， $y = \text{height}$ ， $z = 1.0 + 1.0/\text{width}$ ， $w = 1.0 + 1.0/\text{height}$ ，其中width和height分别是该摄像机的渲染目标（render target）的像素宽度和高度
<code>_ZBufferParams</code>	float4	$x = 1 - \text{Far}/\text{Near}$ ， $y = \text{Far}/\text{Near}$ ， $z = x/\text{Far}$ ， $w = y/\text{Far}$ ，该变量用于线性化Z缓存中的深度值（可参考13.1节）
<code>unity_OrthoParams</code>	float4	$x = \text{width}$ ， $y = \text{height}$ ， $z$ 没有定义， $w = 1.0$ （该摄像机是正交摄像机）或 $w = 0.0$ （该摄像机是透视摄像机），其中width和height是正交投影摄像机的宽度和高度

变量名	类型	描述
unity_CameraProjection	float4x4	该摄像机的投影矩阵
unity_CameraInvProjection	float4x4	该摄像机的投影矩阵的逆矩阵
unity_CameraWorldClipPlanes[6]	float4	该摄像机的6个裁剪平面在世界空间下的等式，按如下顺序：左、右、下、上、近、远裁剪平面

## 4.9 答疑解惑

恭喜你几乎完成了本书所有的数学训练！我们希望你能从上面的内容中得到很多收获和启发。但是，我们也相信在读完上面的内容后你可能对某些概念仍然感到迷惑。不要担心，答疑解惑一节就可以帮你跨过一些障碍。

### 4.9.1 使用 $3\times 3$ 还是 $4\times 4$ 的变换矩阵

对于线性变换（例如旋转和缩放）来说，仅使用 $3\times 3$ 的矩阵就足够表示所有的变换了。但如果存在平移变换，我们就需要使用 $4\times 4$ 的矩阵。因此，在对顶点的变换中，我们通常使用 $4\times 4$ 的变换矩阵。当然，在变换前我们需要把点坐标转换成齐次坐标的表示，即把顶点的 $w$ 分量设为1。而在对方向矢量的变换中，我们通常使用 $3\times 3$ 的矩阵就足够了，这是因为平移变换对方向矢量是没有影响的。

### 4.9.2 Cg中的矢量和矩阵类型



我们通常在Unity Shader中使用Cg作为着色器编程语言。在Cg中变量类型有很多种，但在本节我们是想解释如何使用这些类型进行数学运算。因此，我们只以float家族的变量来做说明。

在Cg中，矩阵类型是由float3x3、float4x4等关键词进行声明和定义的。而对于float3、float4等类型的变量，我们既可以把它当成一个矢量，也可以把它当成是一个 $1 \times n$ 的行矩阵或者一个 $n \times 1$ 的列矩阵。这取决于运算的种类和它们在运算中的位置。例如，当我们进行点积操作时，两个操作数就被当成矢量类型，如下：

```
float4 a = float4(1.0, 2.0, 3.0, 4.0);
float4 b = float4(1.0, 2.0, 3.0, 4.0);
// 对两个矢量进行点积操作
float result = dot(a, b);
```

但在进行矩阵乘法时，参数的位置将决定是按列矩阵还是行矩阵进行乘法。在Cg中，矩阵乘法是通过mul函数实现的。例如：

```
float4 v = float4(1.0, 2.0, 3.0, 4.0);
float4x4 M = float4x4(1.0, 0.0, 0.0, 0.0,
                      0.0, 2.0, 0.0, 0.0,
                      0.0, 0.0, 3.0, 0.0,
                      0.0, 0.0, 0.0, 4.0);
// 把v当成列矩阵和矩阵M进行右乘
float4 column_mul_result = mul(M, v);
// 把v当成行矩阵和矩阵M进行左乘
float4 row_mul_result = mul(v, M);
// 注意: column_mul_result不等于row_mul_result, 而是:
// mul(M,v) == mul(v, transpose(M))
// mul(v,M) == mul(transpose(M), v)
```

因此，参数的位置会直接影响结果值。通常在变换顶点时，我们都是使用右乘的方式来按列矩阵进行乘法。这是因为，Unity提供的内

置矩阵（如UNITY\_MATRIX\_MVP等）都是按列存储的。但有时，我们也会使用左乘的方式，这是因为可以省去对矩阵转置的操作。

需要注意的一点是，Cg对矩阵类型中元素的初始化和访问顺序。在Cg中，对float4x4等类型的变量是按行优先的方式进行填充的。什么意思呢？我们知道，想要填充一个矩阵需要给定一串数字，例如，如果需要声明一个3×4的矩阵，我们需要提供12个数字。那么，这串数字是一行一行地填充矩阵还是一列一列地填充矩阵呢？这两种方式得到的矩阵是不同的。例如，我们使用(1, 2, 3, 4, 5, 6, 7, 8, 9)去填充一个3×3的矩阵，如果是按照行优先的方式，得到的矩阵是：

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

如果是按照列优先的方式，得到的矩阵是：

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

Cg使用的是行优先的方法，即是一行一行地填充矩阵的。因此，如果读者需要自己定义一个矩阵时（例如，自己构建用于空间变换的矩阵），就要注意这里的初始化方式。

类似地，当我们在Cg中访问一个矩阵中的元素时，也是按行来索引的。例如：

```
// 按行优先的方式初始化矩阵M
float3x3 M = float3x3(1.0, 2.0, 3.0,
                      4.0, 5.0, 6.0,
```

```
                7.0, 8.0, 9.0);  
// 得到M的第一行，即(1.0, 2.0, 3.0)  
float3 row = M[0];  
  
// 得到M的第2行第1列的元素，即4.0  
float ele = M[1][0];
```

之所以Unity Shader中的矩阵类型满足上述规则，是因为使用的是Cg语言。换句话说，上面的特性都是Cg的规定。

如果读者熟悉Unity的API，可能知道Unity在脚本中提供了一种矩阵类型——Matrix4x4。脚本中的这个矩阵类型则是采用列优先的方式。这与Unity Shader中的规定不一样，希望读者在遇到时不会感到困惑。

### 4.9.3 Unity中的屏幕坐标: ComputeScreenPos/VPOS/WPOS

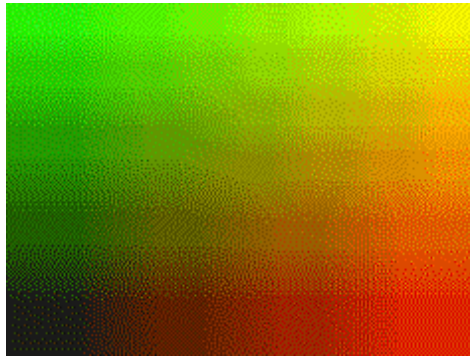
我们在4.6.8节中讲了屏幕空间的转换细节。在写Shader的过程中，我们有时候希望能够获得片元在屏幕上的像素位置。

在顶点/片元着色器中，有两种方式来获得片元的屏幕坐标。

一种是在片元着色器的输入中声明**VPOS**或**WPOS**语义（关于什么是语义，可参见5.4节）。**VPOS**是HLSL中对屏幕坐标的语义，而**WPOS**是Cg中对屏幕坐标的语义。两者在Unity Shader中是等价的。我们可以在HLSL/Cg中通过语义的方式来定义顶点/片元着色器的默认输入，而不需要自己定义输入输出的数据结构。这里的内容有一些超前，因为我们还没有具体讲解顶点/片元着色器的写法，读者在这里可以只关注**VPOS**和**WPOS**的语义。使用这种方法，可以在片元着色器中这样写：

```
fixed4 frag(float4 sp : VPOS) : SV_Target {  
    // 用屏幕坐标除以屏幕分辨率_ScreenParams.xy, 得到视口空间中的坐标  
    return fixed4(sp.xy/_ScreenParams.xy,0.0,1.0);  
}
```

得到的效果如图4.49所示。



▲图4.49 由片元的像素位置得到的图像

VPOS/WPOS语义定义的输入是一个float4类型的变量。我们已经知道它的xy值代表了在屏幕空间中的像素坐标。如果屏幕分辨率为400 x 300, 那么x的范围就是[0.5,400.5], y的范围是[0.5,300.5]。注意, 这里的像素坐标并不是整数值, 这是因为OpenGL和DirectX 10以后的版本认为像素中心对应的是浮点值中的0.5。那么, 它的zw分量是什么呢? 在Unity中, VPOS/WPOS的z分量范围是[0,1], 在摄像机的近裁剪平面处, z值为0, 在远裁剪平面处, z值为1。对于w分量, 我们需要考虑摄像机的投影类型。如果使用的是透视投影, 那么w分量的范围是 $\left[\frac{1}{Near}, \frac{1}{Far}\right]$ , Near和Far对应了在Camera组件中设置的近裁剪平面和远裁剪平面距离摄像机的远近; 如果使用的是正交投影, 那么w分量的值恒为1。这些值是通过经过投影矩阵变换后的w分量取倒数后得到的。在代码的最后, 我们把屏幕空间除以屏幕分辨率来得到**视口空间 (viewport space)** 中的坐标。视口坐标很简单, 就是把屏幕坐标归一

化，这样屏幕左下角就是(0, 0)，右上角就是(1, 1)。如果已知屏幕坐标的话，我们只需要把xy值除以屏幕分辨率即可。

另一种方式是通过Unity提供的**ComputeScreenPos**函数。这个函数在UnityCG.cginc里被定义。通常的用法需要两个步骤，首先在顶点着色器中将ComputeScreenPos的结果保存在输出结构体中，然后在片元着色器中进行一个齐次除法运算后得到视口空间下的坐标。例如：

```
struct vertOut {
    float4 pos:SV_POSITION;
    float4 scrPos : TEXCOORD0;
};

vertOut vert(appdata_base v) {
    vertOut o;
    o.pos = mul (UNITY_MATRIX_MVP, v.vertex);
    // 第一步：把ComputeScreenPos的结果保存到scrPos中
    o.scrPos = ComputeScreenPos(o.pos);
    return o;
}

fixed4 frag(vertOut i) : SV_Target {
    // 第二步：用scrPos.xy除以scrPos.w得到视口空间中的坐标
    float2 wcoord = (i.scrPos.xy/i.scrPos.w);
    return fixed4(wcoord,0.0,1.0);
}
```

上面代码的实现效果和图4.49中的一样。我们现在来看一下这种方式的实现细节。这种方法实际上是手动实现了屏幕映射的过程，而且它得到的坐标直接就是视口空间中的坐标。我们在4.6.8节中已经看到了如何将裁剪坐标空间中的点映射到屏幕坐标中。据此，我们可以得到视口空间中的坐标，公式如下：

$$viewport_x = \frac{clip_x}{2 \cdot clip_w} + \frac{1}{2}$$

上面公式的思想就是，首先对裁剪空间下的坐标进行齐次除法，得到范围在 $[-1, 1]$ 的NDC，然后再将其映射到范围在 $[0, 1]$ 的视口空间下的坐标。那么ComputeScreenPos究竟是如何做到的呢？我们可以在UnityCG.cginc文件中找到ComputeScreenPos函数的定义。如下：

```
inline float4 ComputeScreenPos (float4 pos) {
    float4 o = pos * 0.5f;
    #if defined(UNITY_HALF_TEXEL_OFFSET)
        o.xy = float2(o.x, o.y*_ProjectionParams.x) + o.w *
        _ScreenParams.zw;
    #else
        o.xy = float2(o.x, o.y*_ProjectionParams.x) + o.w;
    #endif

    o.zw = pos.zw;
    return o;
}
```

ComputeScreenPos的输入参数pos是经过MVP矩阵变换后在裁剪空间中的顶点坐标。UNITY\_HALF\_TEXEL\_OFFSET是Unity在某些DirectX平台上使用的宏，在这里我们可以忽略它。这样，我们可以只关注#else的部分。\_ProjectionParams.x在默认情况下是1（如果我们使用了一个翻转的投影矩阵的话就是-1，但这种情况很少见）。那么上述代码的过程实际是输出了：

$$Output_z = clip_z$$

$$Output_w = clip_w$$

可以看出，这里的 $xy$ 并不是真正的视口空间下的坐标。因此，我们在片元着色器中再进行一步处理，即除以裁剪坐标的 $w$ 分量。至此，完成整个映射的过程。因此，虽然`ComputeScreenPos`的函数名字似乎意味着会直接得到屏幕空间中的位置，但并不是这样的，我们仍需在片元着色器中除以它的 $w$ 分量来得到真正的视口空间中的位置。那么，为什么Unity不直接在`ComputeScreenPos`中为我们进行除以 $w$ 分量的这个步骤呢？为什么还需要我们来进行这个除法？这是因为，如果Unity在顶点着色器中这么做的话，就会破坏插值的结果。我们知道，从顶点着色器到片元着色器的过程实际会有一个插值的过程（如果你忘了的话，可以回顾2.3.6小节）。如果不在顶点着色器中进行这个除法，保留 $x$ 、 $y$ 和 $w$ 分量，那么它们在插值后再进行这个除法，得到的 $\frac{x}{w}$ 和 $\frac{y}{w}$ 就是正确的（我们可以认为是除法抵消了插值的影响）。但如果我们直接在顶点着色器中进行这个除法，那么就需要对 $\frac{x}{w}$ 和 $\frac{y}{w}$ 直接进行插值，这样得到的插值结果就会不准确。原因是，我们不可在投影空间中进行插值，因为这并不是一个线性空间，而插值往往是线性的。

经过除法操作后，我们就可以得到该片元在视口空间中的坐标了，也就是一个 $xy$ 范围都在 $[0, 1]$ 之间的值。那么它的 $zw$ 值是什么呢？可以看出，我们在顶点着色器中直接把裁剪空间的 $zw$ 值存进了输出结构体中，因此片元着色器输入的就是这些插值后的裁剪空间中的 $zw$ 值。这意味着，如果使用的是透视投影，那么 $z$ 值的范围是 $[-Near, Far]$ ， $w$ 值的范围是 $[Near, Far]$ ；如果使用的是正交投影，那么 $z$ 值范围是 $[-1, 1]$ ，而 $w$ 值恒为1。

## 4.10 扩展阅读



计算机图形学使用的数学还有很多，本书仅涵盖了其中非常小的一部分。如果读者想要深入学习这些知识的话，书籍<sup>[1][2]</sup>是非常好的图形学数学学习资料，读者可以在那里找到更多类型的变换及其数学表示。关于如何从左手坐标系转换到右手坐标系同时又保持视觉效果一样，可以参考资料<sup>[3]</sup>。关于如何得到线性的深度值可以参考资料<sup>[4]</sup>。

[1] Fletcher Dunn, Ian Parberry. 3D Math Primer for Graphics and Game Development (2nd Edition). November 2, 2011 by A K Peters/CRC Press。

[2] Eric Lengyel. Mathematics for 3D game programming and computer graphics (3rd Edition). 2011 by Charles River Media。

[3] David Eberly. Conversion of Left-Handed Coordinates to Right-Handed Coordinates。

[4] <http://www.humus.name/temp/Linearize%20depth.txt>。

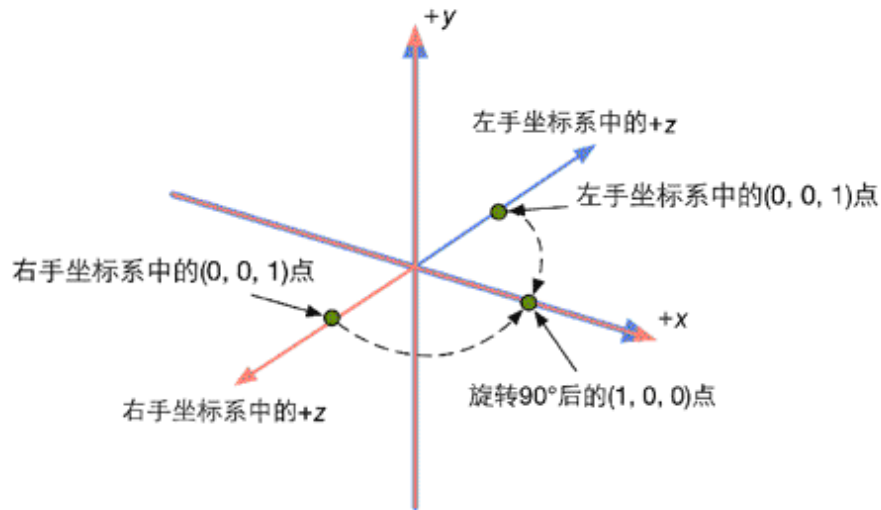
## 4.11 练习题答案

### 4.2.5节

1. 右手坐标系。

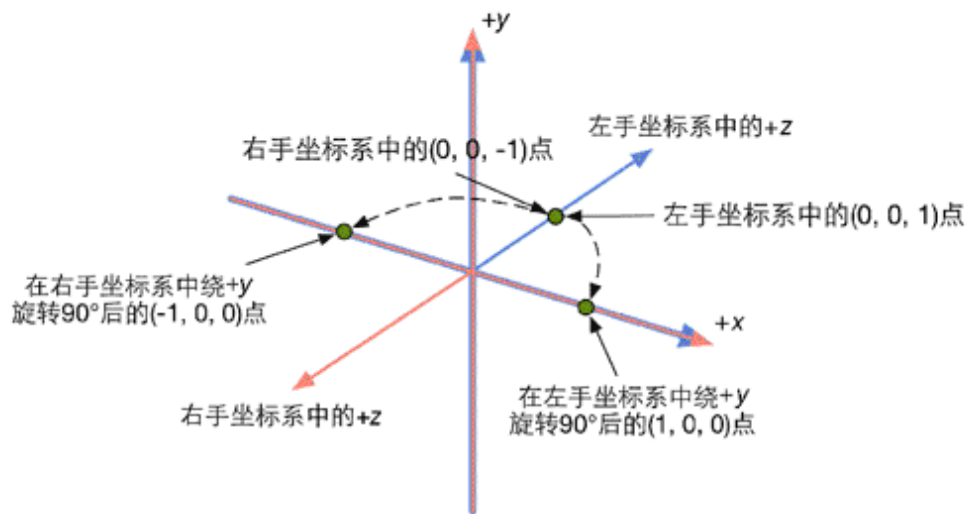
2.  $(1, 0, 0) \circ (1, 0, 0)$ 。从坐标表示来看，结果是完全一样的。左手坐标系和右手坐标系在绝大多数情况下不会对底层的数学运算造成影响，但是会在视觉表现上有所差异。以本题为例，虽然旋转之前点的

坐标是一样的，但如果把它们统一在同一个空间中显示出来，其绝对位置是不同的。如图4.50所示。



▲图4.50 图中两个坐标系的x轴和y轴是重合的，区别仅在于z轴的方向。左手坐标系的(0, 0, 1)点和右手坐标系中的(0, 0, 1)点是不同的，但它们旋转后的点却对应到了同一点

因此，如果我们想要在左手和右手坐标系中表示同一个点，就需要把其中一个坐标系中的表示方法中的某个轴反向，一般是把z值取反。在本例中，左手坐标系的(0, 0, 1)点和右手坐标系中的(0, 0, -1)点是同一点。但是，如果此时对该点再次分别在左手和右手坐标系中绕y轴正方向旋转90°，结果就不是同一个点了，如图4.51所示。



▲图4.51 绝对空间中的同一点，在左手和右手坐标系中进行同样角度的旋转，其旋转方向是不一样的。在左手坐标系中将按顺时针方向旋转，在右手坐标系中将按逆时针方向旋转

3.  $-10^\circ$ 。这是因为，在Unity中，模型空间使用的是左手坐标系。球体所在的位置位于摄像机模型空间中的 $z$ 轴正方向，因此在模型空间下其 $z$ 值为10。而观察空间使用的右手坐标系，摄像机的正前方是 $z$ 轴的负方向，因此在观察空间下其 $z$ 值为 $-10$ 。

### 4.3.3节

1.

(1) 错误，完全说反了。对于矢量来说它有两个属性：模（即大小）和方向，矢量是没有位置属性的，也就是说，我们可以随意把它放在空间的任何位置。

(2) 正确。

(3) 错误。坐标系的选择不会对底层的数学计算产生影响，对于叉积来说，我们总可以使用公式

$$\begin{aligned}\mathbf{a} \times \mathbf{b} &= (a_x, a_y, a_z) \times (b_x, b_y, b_z) \\ &= (a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x)\end{aligned}$$

来计算。但是，不同的坐标系会影响最后的显示结果，即视觉上的表现。数学是一门非常严谨的学科，但人类往往需要可视化一些东西，例如在屏幕上显示虚拟的三维空间，在把数字转换成视觉表现的时候，选择不同的坐标系可能会得到不同的结果。

2.

(1)  $\sqrt{62} \approx 7.874$

(2) (12.5,10,25)

(3) (1.5,2)

(4)  $\left(\frac{5}{13} \frac{12}{13}\right) \approx 0.3850.923$

(5)  $\left(\frac{1}{\sqrt{3}} \frac{1}{\sqrt{3}} \frac{1}{\sqrt{3}}\right) \approx 0.5770.5770.577$

(6) (10,9)

(7) (-6,1,2)

3.  $\sqrt{308} \approx 17.55$

4.

(1) 75

(2) 13

(3) 13

(4) (-9,-13,7)

(5) (9,13,-7), 注意, 结果和答案 (4) 是相反的。这是因为, 叉积满足反交换律。

5.

(1) 12

(2)  $12\sqrt{3} \approx 20.785$

6.

(1) 我们可以通过判断 $\mathbf{x-p}$ 和 $\mathbf{v}$ 点积的符号来判断 $\mathbf{x}$ 是否在NPC的前方。这是因为:

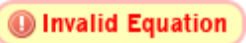
$$(\mathbf{x-p}) \cdot \mathbf{v} = |\mathbf{v}| |\mathbf{x-p}| \cos \theta$$

其中 $\theta$ 是 $\mathbf{x-p}$ 和 $\mathbf{v}$ 之间的夹角。如果它们点积的结果大于0, 那么说明 $\theta < 90^\circ$ , 即点 $\mathbf{x}$ 在NPC的前方; 如果点积结果小于0, 那么说明 $\theta > 90^\circ$ , 即点 $\mathbf{x}$ 在NPC的后方; 如果点积结果等于0, 那么说明 $\theta = 90^\circ$ , 即点 $\mathbf{x}$ 在NPC的正左侧或正右侧。

(2) 代入得

$$(\mathbf{x}-\mathbf{p})\cdot\mathbf{v}=((10,6)-(4,2))\cdot(-3,4)=(6,4)\cdot(-3,4)=-18+16=-2<0$$

因此，点 $\mathbf{x}$ 在NPC的后方。

(3) 我们现在需要判断 $\cos\theta$ 和 $\cos\frac{\phi}{2}$ 的大小。如果 , 那么说明，即NPC可以看到该点；如果，那么说明，即NPC无法看到该点。 $\cos\theta$ 可以由  $\cos\theta = \frac{(\mathbf{x}-\mathbf{p})\cdot\mathbf{v}}{|\mathbf{x}-\mathbf{p}||\mathbf{v}|}$  来得到，而可直接计算得到。

(4) 如果有距离限制，我们只需要判断该点到 $\mathbf{p}$ 的距离是否小于该限制值即可。

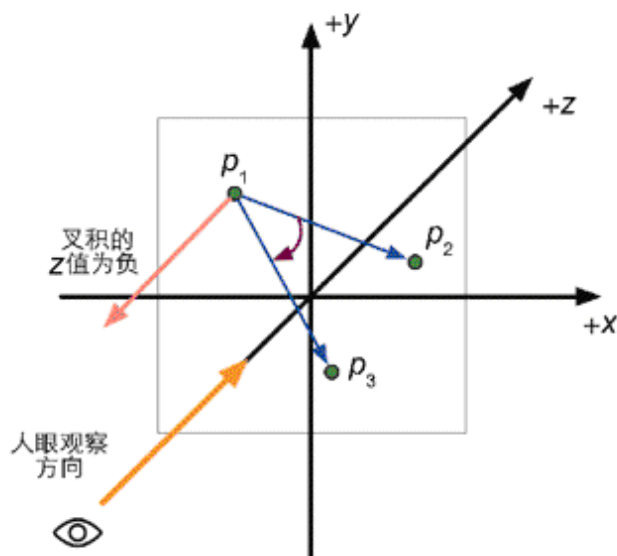
7. 令 $\mathbf{u}=\mathbf{p}_2-\mathbf{p}_1$ ， $\mathbf{v}=\mathbf{p}_3-\mathbf{p}_1$ 。由于三点都位于 $xy$ 平面，那么有：

$$\mathbf{u}=(u_x, u_y, 0), \mathbf{v}=(v_x, v_y, 0)$$

它们的叉积为：

$$\mathbf{u}\times\mathbf{v}=(0,0,u_xv_y-u_yv_x)$$

我们可以通过判断 $u_xv_y-u_yv_x$ 的符号来判断三角形的朝向。如果该值为负，则由左手法则判断可得到3个顶点的顺序是顺时针方向；如果为正，则为逆时针方向。如图4.52所示。



▲图4.52 在左手坐标系中，如果叉积结果为负，那么3点的顺序是顺时针方向

#### 4.4.6小节

1.

$$(1) \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} -1 & 5 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} (1)(-1) + (3)(0) & (1)(5) + (3)(2) \\ (2)(-1) + (4)(0) & (2)(5) + (4)(2) \end{bmatrix} = \begin{bmatrix} -1 & 11 \\ -2 & 18 \end{bmatrix}$$

(2) 无法进行矩阵乘法，两个矩阵相乘要求第一个矩阵的列数等于第二个的行数，因此我们无法对 $2 \times 3$ 和 $4 \times 2$ 的矩阵进行乘法。

$$(3) \begin{bmatrix} 1 & -2 & 3 \\ 5 & 1 & 4 \\ 6 & 0 & 3 \end{bmatrix} \begin{bmatrix} -5 \\ 4 \\ 8 \end{bmatrix} = \begin{bmatrix} (1)(-5) + (-2)(4) + (3)(8) \\ (5)(-5) + (1)(4) + (4)(8) \\ (6)(-5) + (0)(4) + (3)(8) \end{bmatrix} = \begin{bmatrix} 11 \\ 11 \\ -6 \end{bmatrix}$$

2.

(1) 不是正交矩阵。它的转置矩阵和本身相乘的结果不是单位矩阵。也可以通过验证矩阵的行是否构成一组标准正交基来判断。



(2) 是正交矩阵。

(3) 是正交矩阵。这实际上是一个绕z轴旋转 $\theta^\circ$ 的旋转矩阵。

3.

$$(1) \quad [3 \quad 2 \quad 6] \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} (3)(1) + (2)(0) + (6)(0) \\ (3)(0) + (2)(1) + (6)(0) \\ (3)(0) + (2)(0) + (6)(1) \end{bmatrix} = [3 \quad 2 \quad 6]$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 2 \\ 6 \end{bmatrix} = \begin{bmatrix} (1)(3) + (0)(2) + (0)(6) \\ (0)(3) + (1)(2) + (0)(6) \\ (0)(3) + (0)(2) + (1)(6) \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \\ 6 \end{bmatrix}$$

得到的结果转换成矢量都是(3,2,6)，是一样的。这是因为，该矩阵是一个单位矩阵，单位矩阵和任何矩阵相乘都是原矩阵本身。

$$(2) \quad [3 \quad 2 \quad 6] \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & -3 \\ 0 & 0 & 3 \end{bmatrix} = \begin{bmatrix} (3)(1) + (2)(0) + (6)(0) \\ (3)(0) + (2)(1) + (6)(0) \\ (3)(2) + (2)(-3) + (6)(3) \end{bmatrix} = [3 \quad 2 \quad 18]$$

$$\begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & -3 \\ 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} 3 \\ 2 \\ 6 \end{bmatrix} = \begin{bmatrix} (1)(3) + (0)(2) + (2)(6) \\ (0)(3) + (1)(2) + (-3)(6) \\ (0)(3) + (0)(2) + (3)(6) \end{bmatrix} = \begin{bmatrix} 15 \\ -16 \\ 18 \end{bmatrix}$$

得到的结果不一致。为了得到一致的结果，我们可以对矩阵进行转置。例如，为了得到和列矩阵相同的结果，在进行行矩阵乘法时，对矩阵进行转置，得

$$\begin{aligned}
& [3 \quad 2 \quad 6] \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & -3 \\ 0 & 0 & 3 \end{bmatrix}^T = [3 \quad 2 \quad 6] \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 2 & -3 & 3 \end{bmatrix} \\
& = \begin{bmatrix} (3)(1) + (2)(0) + (6)(2) \\ (3)(0) + (2)(1) + (6)(-3) \\ (3)(0) + (2)(0) + (6)(3) \end{bmatrix} = [15 \quad -16 \quad 18]
\end{aligned}$$

$$(3) \quad [3 \quad 2 \quad 6] \begin{bmatrix} 2 & -1 & 3 \\ -1 & 5 & -3 \\ 3 & -3 & 4 \end{bmatrix} = \begin{bmatrix} (3)(2) + (2)(-1) + (6)(3) \\ (3)(-1) + (2)(5) + (6)(-3) \\ (3)(3) + (2)(-3) + (6)(4) \end{bmatrix} = [22 \quad -11 \quad 27]$$

$$\begin{bmatrix} 2 & -1 & 3 \\ -1 & 5 & -3 \\ 3 & -3 & 4 \end{bmatrix} \begin{bmatrix} 3 \\ 2 \\ 6 \end{bmatrix} = \begin{bmatrix} (2)(3) + (-1)(2) + (3)(6) \\ (-1)(3) + (5)(2) + (-3)(6) \\ (3)(3) + (-3)(2) + (4)(6) \end{bmatrix} = \begin{bmatrix} 22 \\ -11 \\ 27 \end{bmatrix}$$

得到的结果转换成矢量都是(22,-11,27)，是一样的。这是因为，该矩阵是一个**对称矩阵**（**symmetric matrix**）。对称矩阵的转置是其本身，因此行矩阵和列矩阵不会对结果产生影响。

## 第2篇 初级篇

在学习完基础篇后，我们就正式开始了Unity Shader的学习之旅。初级篇将会从最简单的Shader开始，讲解Shader中基础的光照模型、纹理和透明效果等初级渲染效果。需要注意的是，我们在初级篇中实现的Unity Shader大多不能直接用于真实项目中，因为它们缺少了完整的光照计算，例如阴影、光照衰减等，仅仅是为了阐述一些实现原理。在第9章，会给出包含了完整光照计算的Unity Shader。

### 第5章 开始Unity Shader学习之旅

本章将实现一个简单的顶点/片元着色器，并详细解释其中每个步骤的原理，还会给出关于Unity Shader的一些常用的辅助技巧等。

### 第6章 Unity中的基础光照

本章将学习如何在Shader中实现基本的光照模型，如漫反射、高光反射等。

### 第7章 基础纹理

这一章将会讲述如何在Unity Shader中使用法线纹理、遮罩纹理等基础纹理。

### 第8章 透明效果

这一章首先介绍了渲染的实现原理，并给出了和Unity的渲染顺序相关的重要内容。在了解了这些内容的基础上，我们将学习如何实现透明度测试和透明度混合等透明效果。

## 第5章 开始Unity Shader学习之旅

欢迎来到本书的第2篇——初级篇。在基础篇中，我们学习了渲染流水线，并给出了Unity Shader的基本概况，同时还打下了一定的数学基础。从本章开始，我们将真正开始学习如何在Unity中编写Unity Shader。

本章的结构如下：在5.1节，我们将给出编写本书时使用的软件，包括Unity的版本等。这是为了让读者可以在实践时不会出现因版本不同而造成困扰。在5.2节，我们将看到一个最简单的顶点/片元着色器，并详细地解释这个顶点/片元着色器的组成结构。5.3节将介绍Unity内置的Unity Shader文件，以及提供给用户的一些包含文件、内置变量和函数等。5.4节则向读者阐述Unity Shader中使用的Cg语义，这是很多初学者容易困惑的地方。在5.5节中，我们会介绍如何对Unity Shader进行调试。5.6节将介绍平台差异对Unity Shader的影响。最后，5.7节将给出一些在编写Unity Shader时很容易实现的优化技巧。为了让读者养成良好的编程习惯，我们在这节也给出了一些建议。

### 5.1 本书使用的软件和环境

本书使用的Unity版本是Unity 5.2.1免费版。使用更高版本的Unity通常不会有什么影响。但如果你打算使用更低版本的Unity，那么在学习本书时可能就会遇到一些问题。例如，你发现有些菜单或变量在你安装的Unity中找不到，可能就是因为Unity版本不同造成的。绝大多数

情况下，本书的代码和指令仍然可以工作良好，但在一些特殊情况下，Unity可能会更改底层的实现细节，造成同样的代码得到不一样的效果（例如，在非统一缩放时对法线进行变换，详见19.3节）。还有一些问题是Unity提供的内置变量、宏和函数，例如我们在书中经常会使用UnityObjectToWorldNormal内置函数把法线从模型空间变换到世界空间中，但这个函数是在Unity 5中被引入的，因此如果读者使用的是Unity 5之前的版本就会报错。类似的情况还有和阴影相关的宏和变量等。和Unity 4.x版本相比，Unity 5.x最大的变化之一就是很多以前只有在专业版才支持的功能，在免费版也同样提供了。因此，如果读者使用的是Unity 4.x免费版，可能会发现本书中的某些示例无法实现。

本书工程编写的系统环境是Mac OS X 10.9.5。如果读者使用的是其他系统，绝大部分情况也不会有任何问题。但有时会由于图像编程接口的种类和版本不同而有一些差别，这是因为Mac使用的图像编程接口是基于OpenGL的，而其他平台如Windows，可能使用的是DirectX。例如，在OpenGL中，渲染纹理（Render Texture）的(0, 0)点是在左下角，而在DirectX中，(0, 0)点是在左上角。在5.6节，我们将总结一些由于平台而造成的差异问题。

## 5.2 一个最简单的顶点/片元着色器

现在，我们正式开始学习如何编写Unity Shader，更准确地说是，学习如何编写顶点/片元着色器。

### 5.2.1 顶点/片元着色器的基本结构

我们在3.3节已经看到了Unity Shader的基本结构。它包含了 *Shader*、*Properties*、*SubShader*、*Fallback*等语义块。顶点/片元着色器的结构与之大体类似，它的结构如下：

```
Shader "MyShaderName" {
    Properties {
        // 属性
    }
    SubShader {
        // 针对显卡A的SubShader
        Pass {
            // 设置渲染状态和标签

            // 开始Cg代码片段
            CGPROGRAM
            // 该代码片段的编译指令，例如：
            #pragma vertex vert
            #pragma fragment frag

            // Cg代码写在这里

            ENDCG

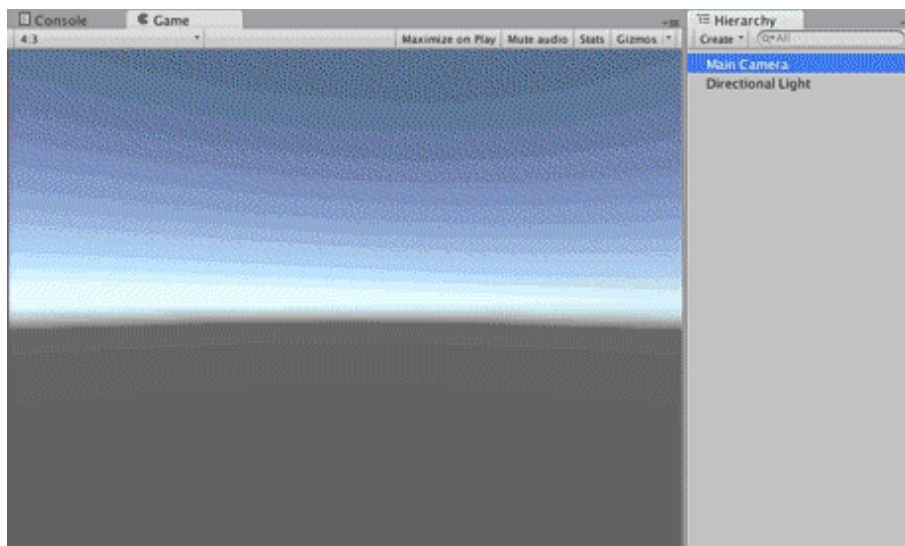
            // 其他设置
        }
        // 其他需要的Pass
    }
    SubShader {
        // 针对显卡B的SubShader
    }

    // 上述SubShader都失败后用于回调的Unity Shader
    Fallback "VertexLit"
}
```

其中，最重要的部分是*Pass*语义块，我们绝大部分的代码都是写在这个语义块里面的。下面我们就来创建一个最简单的顶点/片元着色器。

(1) 新建一个场景，把它命名为Scene\_5\_2。在Unity 5中可以得到图5.1中的效果。





▲图5.1 在Unity 5中新建一个场景得到的效果

可以看到，场景中已经包含了一个摄像机、一个平行光。而且，场景的背景不是纯色，而是一个天空盒子（Skybox）。这是因为在Unity 5.x版本中，默认的天空盒子不为空，而是Unity内置的一个天空盒子。为了得到更加原始的效果，我们选择去掉这个天空盒子。做法是，在Unity的菜单中，选择Window -> Lighting -> Skybox，把该项置为空。注意，在Unity 4.x版本中，设置天空盒子的位置与这里并不一样。

（2）新建一个Unity Shader，把它命名为Chapter5-SimpleShader。

（3）新建一个材质，把它命名为SimpleShaderMat。把第2步中新建的Unity Shader赋给它。

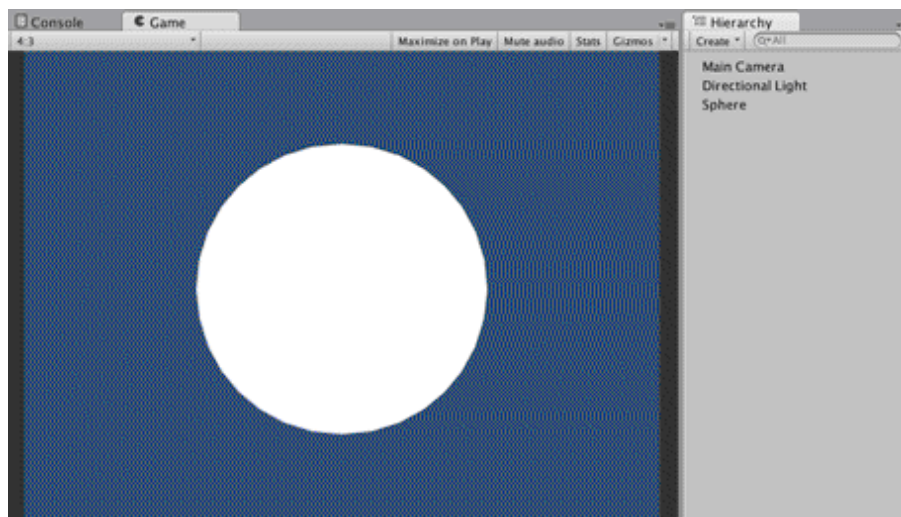
（4）新建一个球体，拖曳它的位置以便在Game视图中可以合适地显示出来。把第3步中新建的材质拖曳给它。

(5) 双击打开第2步中创建的Unity Shader。删除里面的所有代码，把下面的代码粘贴进去：

```
Shader "Unity Shaders Book/Chapter 5/Simple Shader" {  
    SubShader {  
        Pass {  
            CGPROGRAM  
  
            #pragma vertex vert  
            #pragma fragment frag  
  
            float4 vert(float4 v : POSITION) : SV_POSITION {  
                return mul (UNITY_MATRIX_MVP, v);  
            }  
  
            fixed4 frag() : SV_Target {  
                return fixed4(1.0, 1.0, 1.0, 1.0);  
            }  
  
            ENDCG  
        }  
    }  
}
```

保存并返回Unity查看结果。

最后，我们得到的结果如图5.2所示。



▲图5.2 用一个最简单的顶点/片元着色器得到一个白色的球

这是我们遇见的第一个真正意义上的顶点/片元着色器，我们有必要来详细地解释一下它。

首先，代码的第一行通过`Shader`语义定义了这个Unity Shader的名字——“Unity Shaders Book/Chapter 5/Simple Shader”。保持良好的命名习惯有助于我们在为材质球选择`Shader`时快速找到自定义的Unity `Shader`。需要注意的是，在上面的代码里我们并没有用到`Properties`语义块。`Properties`语义并不是必需的，我们可以选择不声明任何材质属性。

然后，我们声明了`SubShader`和`Pass`语义块。在本例中，我们不需要进行任何渲染设置和标签设置，因此`SubShader`将使用默认的渲染设置和标签设置。在`SubShader`语义块中，我们定义了一个`Pass`，在这个`Pass`中我们同样没有进行任何自定义的渲染设置和标签设置。

接着，就是由`CGPROGRAM`和`ENDCG`所包围的CG代码片段。这是我们的重点。首先，我们遇到了两行非常重要的编译指令：

```
#pragma vertex vert
#pragma fragment frag
```

它们将告诉Unity，哪个函数包含了顶点着色器的代码，哪个函数包含了片元着色器的代码。更通用的编译指令表示如下：

```
#pragma vertex name
#pragma fragment name
```

其中**name**就是我们指定的函数名，这两个函数的名字不一定是**vert**和**frag**，它们可以是任意自定义的合法函数名，但我们一般使用**vert**和**frag**来定义这两个函数，因为它们很直观。

接下来，我们具体看一下**vert**函数的定义：

```
float4 vert(float4 v : POSITION) : SV_POSITION {  
    return mul (UNITY_MATRIX_MVP, v);  
}
```

这就是本例使用的顶点着色器代码，它是逐顶点执行的。**vert**函数的输入**v**包含了这个顶点的位置，这是通过**POSITION**语义指定的。它的返回值是一个**float4**类型的变量，它是该顶点在裁剪空间中的位置，**POSITION**和**SV\_POSITION**都是Cg/HLSL中的**语义（semantics）**，它们是不可省略的，这些语义将告诉系统用户需要哪些输入值，以及用户的输出是什么。例如这里，**POSITION**将告诉Unity，把模型的顶点坐标填充到输入参数**v**中，**SV\_POSITION**将告诉Unity，顶点着色器的输出是裁剪空间中的顶点坐标。如果没有这些语义来限定输入和输出参数的话，渲染器就完全不知道用户的输入输出是什么，因此就会得到错误的效果。在5.4节中，我们将总结这些语义。在本例中，顶点着色器只包含了一行代码，这行代码读者应该已经很熟悉了（起码对这个数学操作应该很熟悉了），这一步就是把顶点坐标从模型空间转换到裁剪空间中。**UNITY\_MATRIX\_MVP**矩阵是Unity内置的模型·观察·投影矩阵，我们在4.8节已经见过它了。

然后，我们再来看一下**frag**函数：

```
fixed4 frag() : SV_Target {  
    return fixed4(1.0, 1.0, 1.0, 1.0);  
}
```

在本例中，`frag`函数没有任何输入。它的输出是一个`fixed4`类型的变量，并且使用了`SV_Target`语义进行限定。`SV_Target`也是HLSL中的一个系统语义，它等同于告诉渲染器，把用户的输出颜色存储到一个渲染目标（`render target`）中，这里将输出到默认的帧缓存中。片元着色器中的代码很简单，返回了一个表示白色的`fixed4`类型的变量。片元着色器输出的颜色的每个分量范围在 $[0, 1]$ ，其中 $(0, 0, 0)$ 表示黑色，而 $(1, 1, 1)$ 表示白色。

至此，我们已经对第一个顶点/片元着色器进行了详细的解释。但是，现在得到的效果实在是太简单了，如何丰富它呢？下面我们将一步步为它添加更多的内容，以得到一个更加具有实践意义的顶点/片元着色器。

## 5.2.2 模型数据从哪里来

在上面的例子中，在顶点着色器中我们使用`POSITION`语义得到了模型的顶点位置。那么，如果我们想要得到更多模型数据怎么办呢？

现在，我们想要得到模型上每个顶点的纹理坐标和法线方向。这个需求是很常见的，我们需要使用纹理坐标来访问纹理，而法线可用于计算光照。因此，我们需要为顶点着色器定义一个新的输入参数，这个参数不再是一个简单的数据类型，而是一个结构体。修改后的代码如下：

```
Shader "Unity Shaders Book/Chapter 5/Simple Shader" {
    SubShader {
        Pass {
            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag
```

```

// 使用一个结构体来定义顶点着色器的输入
struct a2v {
    // POSITION语义告诉Unity，用模型空间的顶点坐标填充vertex
    float4 vertex : POSITION;
    // NORMAL语义告诉Unity，用模型空间的法线方向填充normal变量
    float3 normal : NORMAL;
    // TEXCOORD0语义告诉Unity，用模型的第一套纹理坐标填充texcoord变量
    float4 texcoord : TEXCOORD0;
};

float4 vert(a2v v) : SV_POSITION {
    // 使用v.vertex来访问模型空间的顶点坐标
    return mul (UNITY_MATRIX_MVP, v.vertex);
}

fixed4 frag() : SV_Target {
    return fixed4(1.0, 1.0, 1.0, 1.0);
}

ENDCG
}
}
}

```

在上面的代码中，我们声明了一个新的结构体**a2v**，它包含了顶点着色器需要的模型数据。在**a2v**的定义中，我们用到了更多Unity支持的语义，如**NORMAL**和**TEXCOORD0**，当它们作为顶点着色器的输入时都是有特定含义的，因为Unity会根据这些语义来填充这个结构体。对于顶点着色器的输入，Unity支持的语义有：*POSITION*, *TANGENT*, *NORMAL*, *TEXCOORD0*, *TEXCOORD1*, *TEXCOORD2*, *TEXCOORD3*, *COLOR*等。

为了创建一个自定义的结构体，我们必须使用如下格式来定义它：

```
struct StructName {  
    Type Name : Semantic;  
    Type Name : Semantic;  
    .....  
};
```

其中，语义是不可以被省略的。在5.4节中，我们将给出这些语义的含义和用法。

然后，我们修改了`vert`函数的输入参数类型，把它设置为我们新定义的结构体`a2v`。通过这种自定义结构体的方式，我们就可以在顶点着色器中访问模型数据。

读者：`a2v`的名字是什么意思呢？

我们：`a`表示应用（`application`），`v`表示顶点着色器（`vertex shader`），`a2v`的意思就是把数据从应用阶段传递到顶点着色器中。

那么，填充到`POSITION`，`TANGENT`，`NORMAL`这些语义中的数据究竟是从哪里来的呢？在Unity中，它们是由使用该材质的`Mesh Render`组件提供的。在每帧调用`Draw Call`的时候，`Mesh Render`组件会把它负责渲染的模型数据发送给`Unity Shader`。我们知道，一个模型通常包含了一组三角面片，每个三角面片由3个顶点构成，而每个顶点又包含了一些数据，例如顶点位置、法线、切线、纹理坐标、顶点颜色等。通过上面的方法，我们就可以在顶点着色器中访问顶点的这些模型数据。

### 5.2.3 顶点着色器和片元着色器之间如何通信



在实践中，我们往往希望从顶点着色器输出一些数据，例如把模型的法线、纹理坐标等传递给片元着色器。这就涉及顶点着色器和片元着色器之间的通信。

为此，我们需要再定义一个新的结构体。修改后的代码如下：

```
Shader "Unity Shaders Book/Chapter 5/Simple Shader" {
    SubShader {
        Pass {
            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            struct a2v {
                float4 vertex : POSITION;
                float3 normal : NORMAL;
                float4 texcoord : TEXCOORD;
            };

            // 使用一个结构体来定义顶点着色器的输出
            struct v2f {
                // SV_POSITION语义告诉Unity，pos里包含了顶点在裁剪空间中
                的位置信息
                float4 pos : SV_POSITION;
                // COLOR0语义可以用于存储颜色信息
                fixed3 color : COLOR0;
            };

            v2f vert(a2v v) {
                // 声明输出结构
                v2f o;
                o.pos = mul(UNITY_MATRIX_MVP, v.vertex);
                // v.normal包含了顶点的法线方向，其分量范围在[-1.0, 1.0]
                // 下面的代码把分量范围映射到了[0.0, 1.0]
                // 存储到o.color中传递给片元着色器
                o.color = v.normal * 0.5 + fixed3(0.5, 0.5, 0.5);
                return o;
            }

            fixed4 frag(v2f i) : SV_Target {
                // 将插值后的i.color显示到屏幕上
                return fixed4(i.color, 1.0);
            }

            ENDCG
        }
    }
}
```

```
}  
}  
}
```

在上面的代码中，我们声明了一个新的结构体`v2f`。`v2f`用于在顶点着色器和片元着色器之间传递信息。同样的，`v2f`中也需要指定每个变量的语义。在本例中，我们使用了`SV_POSITION`和`COLOR0`语义。顶点着色器的输出结构中，必须包含一个变量，它的语义是`SV_POSITION`。否则，渲染器将无法得到裁剪空间中的顶点坐标，也就无法把顶点渲染到屏幕上。`COLOR0`语义中的数据则可以由用户自行定义，但一般都是存储颜色，例如逐顶点的漫反射颜色或逐顶点的高光反射颜色。类似的语义还有`COLOR1`等，具体可以详见5.4节。

至此，我们就完成了顶点着色器和片元着色器之间的通信。需要注意的是，顶点着色器是逐顶点调用的，而片元着色器是逐片元调用的。片元着色器中的输入实际上是把顶点着色器的输出进行插值后得到的结果。

## 5.2.4 如何使用属性

在3.1.1节中，我们就提到了材质和Unity Shader之间的紧密联系。材质提供给我们一个可以方便地调节Unity Shader中参数的方式，通过这些参数，我们可以随时调整材质的效果。而这些参数就需要写在Properties语义块中。

现在，我们有了新的需求。我们想要在材质面板显示一个颜色拾取器，从而可以直接控制模型在屏幕上显示的颜色。为此，我们继续修改上面的代码。

```

Shader "Unity Shaders Book/Chapter 5/Simple Shader" {
    Properties {
        // 声明一个Color类型的属性
        _Color ("Color Tint", Color) = (1.0,1.0,1.0,1.0)
    }
    SubShader {
        Pass {
            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            // 在Cg代码中，我们需要定义一个与属性名称和类型都匹配的变量
            fixed4 _Color;

            struct a2v {
                float4 vertex : POSITION;
                float3 normal : NORMAL;
                float4 texcoord : TEXCOORD0;
            };

            struct v2f {
                float4 pos : SV_POSITION;
                fixed3 color : COLOR0;
            };

            v2f vert(a2v v) {
                v2f o;
                o.pos = mul(UNITY_MATRIX_MVP, v.vertex);
                o.color = v.normal * 0.5 + fixed3(0.5, 0.5, 0.5);
                return o;
            }

            fixed4 frag(v2f i) : SV_Target {
                fixed3 c = i.color;
                // 使用_Color属性来控制输出颜色
                c *= _Color.rgb;
                return fixed4(c, 1.0);
            }

            ENDCG
        }
    }
}

```

在上面的代码中，我们首先添加了*Properties*语义块中，并在其中声明了一个属性\_Color，它的类型是Color，初始值是(1.0,1.0,1.0,1.0)，

对应白色。为了在Cg代码中可以访问它，我们还需要在Cg代码片段中提前定义一个新的变量，这个变量的名称和类型必须与*Properties*语义块中的属性定义相匹配。

ShaderLab中属性的类型和Cg中变量的类型之间的匹配关系如表5.1所示。

表5.1 ShaderLab属性类型和Cg变量类型的匹配关系

ShaderLab属性类型	Cg变量类型
Color, Vector	float4, half4, fixed4
Range, Float	float, half, fixed
2D	sampler2D
Cube	samplerCube
3D	sampler3D

有时，读者可能会发现在Cg变量前会有一个uniform关键字，例如：

```
uniform fixed4 _Color;
```

uniform关键词是Cg中修饰变量和参数的一种修饰词，它仅仅用于提供一些关于该变量的初始值是如何指定和存储的相关信息（这和其

他一些图像编程接口中的uniform关键词的作用不太一样)。在Unity Shader中，uniform关键词是可以省略的。

## 5.3 强大的援手：Unity提供的内置文件和变量

上一节讲述了如何在Unity中编写一个基本的顶点/片元着色器的过程。顶点/片元着色的复杂之处在于，很多事情都需要我们“亲力亲为”，例如我们需要自己转换法线方向，自己处理光照、阴影等。为了方便开发者的编码过程，Unity提供了很多内置文件，这些文件包含了很多提前定义的函数、变量和宏等。如果读者在学习他人编写的Unity Shader代码时，遇到了一些从未见过的变量、函数，而又无法找到对应的声明和定义，那么很有可能就是这些代码使用了Unity内置文件提供的函数和变量。

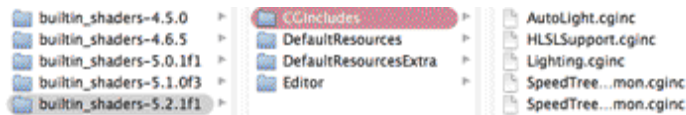
本节将给出这些文件和变量的概览。

### 5.3.1 内置的包含文件

**包含文件 (include file)**，是类似于C++中头文件的一种文件。在Unity中，它们的文件后缀是.cginc。在编写Shader时，我们可以使用#include指令把这些文件包含进来，这样我们就可以使用Unity为我们提供的一些非常有用的变量和帮助函数。例如：

```
CGPROGRAM
// ...
#include "UnityCG.cginc"
// ...
ENDCG
```

那么，这些文件在哪里呢？我们可以在官方网站（<http://unity3d.com/cn/get-unity/download/archive>）上选择下载 -> 内置着色器来直接下载这些文件，图5.3显示了由官网压缩包得到的文件。



▲图5.3 Unity的内置着色器

从图5.3中可以看出，从官网下载的文件中包含了多个文件夹。其中，CGIncludes文件夹中包含了所有的内置包含文件；DefaultResources文件夹中包含了一些内置组件或功能所需要的Unity Shader，例如一些GUI元素使用的Shader；DefaultResourcesExtra则包含了所有Unity中内置的Unity Shader；Editor文件夹目前只包含了一个脚本文件，它用于定义Unity 5引入的Standard Shader（详见第18章）所用的材质面板。这些文件都是非常好的参考资料，在我们想要学习内置着色器的实现或是寻找内置函数的实现时，都可以在这里找到内部实现。但在本节中，我们只关注CGIncludes文件夹下的相关文件。

我们也可以从Unity的应用程序中直接找到CGIncludes文件夹。在Mac上，它们的位置是：/Applications/Unity/Unity.app/Contents/CGIncludes；在Windows上，它们的位置是：Unity的安装路径/Data/CGIncludes。

表5.2给出了CGIncludes中主要的包含文件以及它们的主要用处。

表5.2 Unity中一些常用的包含文件

文 件 名	描 述
UnityCG.cginc	包含了最常使用的帮助函数、宏和结构体等
UnityShaderVariables.cginc	在编译Unity Shader时，会被自动包含进来。包含了许多内置的全局变量，如UNITY_MATRIX_MVP等
Lighting.cginc	包含了各种内置的光照模型，如果编写的是Surface Shader的话，会自动包含进来
HLSLSupport.cginc	在编译Unity Shader时，会被自动包含进来。声明了很多用于跨平台编译的宏和定义

可以看出，有一些文件是即便我们没有使用`#include`指令，它们也是会被自动包含进来的，例如`UnityShaderVariables.cginc`。因此，在前面的例子中，我们可以直接使用`UNITY_MATRIX_MVP`变量来进行顶点变换。除了表5.2中列出的包含文件外，Unity 5引入了许多新的重要的包含文件，如`UnityStandardBRDF.cginc`、`UnityStandardCore.cginc`等，这些包含文件用于实现基于物理的渲染，我们会在第18章中再次遇到它们。

`UnityCG.cginc`是我们最常接触的一个包含文件。在后面的学习中，我们将使用很多该文件提供的结构体和函数，为我们的编写提供方便。例如，我们可以直接使用`UnityCG.cginc`中预定义的结构体作为顶点着色器的输入和输出。表5.3给出了一些结构体的名称和包含的变量。



**表5.3      UnityCG.cginc中一些常用的结构体**

名 称	描 述	包含的变量
appdata_base	可用于顶点着色器的输入	顶点位置、顶点法线、第一组纹理坐标
appdata_tan	可用于顶点着色器的输入	顶点位置、顶点切线、顶点法线、第一组纹理坐标
appdata_full	可用于顶点着色器的输入	顶点位置、顶点切线、顶点法线、四组（或更多）纹理坐标
appdata_img	可用于顶点着色器的输入	顶点位置、第一组纹理坐标
v2f_img	可用于顶点着色器的输出	裁剪空间中的位置、纹理坐标

强烈建议读者找到UnityCG.cginc文件并查看上述结构体的声明，这样的过程可以帮助我们快速理解Unity中一些内置变量的工作原理。

除了结构体外，UnityCG.cginc也提供了一些常用的帮助函数。表5.4给出了一些函数名和它们的描述。

**表5.4      UnityCG.cginc中一些常用的帮助函数**

函 数 名	描 述
-------	-----

函 数 名	描 述
float3 WorldSpaceViewDir (float4 v)	输入一个模型空间中的顶点位置，返回世界空间中从该点到摄像机的观察方向
float3 ObjSpaceViewDir (float4 v)	输入一个模型空间中的顶点位置，返回模型空间中从该点到摄像机的观察方向
float3 WorldSpaceLightDir (float4 v)	<b>仅可用于前向渲染中。</b> 输入一个模型空间中的顶点位置，返回世界空间中从该点到光源的光照方向。没有被归一化
float3 ObjSpaceLightDir (float4 v)	<b>仅可用于前向渲染中。</b> 输入一个模型空间中的顶点位置，返回模型空间中从该点到光源的光照方向。没有被归一化
float3 UnityObjectToWorldNormal (float3 norm)	把法线方向从模型空间转换到世界空间中
float3 UnityObjectToWorldDir (float3 dir)	把方向矢量从模型空间变换到世界空间中
float3 UnityWorldToObjectDir(float3 dir)	把方向矢量从世界空间变换到模型空间中

我们建议读者在UnityCG.cginc文件找到这些函数的定义，并尝试理解它们。一些函数我们完全可以自己实现，例如UnityObjectToWorldDir和UnityWorldToObjectDir，这两个函数实际上就是对方向矢量进行了一次坐标空间变换。而UnityCG.cginc文件可以帮助我们提高代码的复用率。UnityCG.cginc还包含了很多宏，在后面的学习中，我们就会遇到它们。

### 5.3.2 内置的变量

我们在4.8节给出了一些用于坐标变换和摄像机参数的内置变量。除此之外，Unity还提供了用于访问时间、光照、雾效和环境光等目的的变量。这些内置变量大多位于UnityShader Variables.cginc中，与光照有关的内置变量还会位于Lighting.cginc、AutoLight.cginc等文件中。当我们在后面的学习中遇到这些变量时，再进行详细的讲解。

## 5.4 Unity提供的Cg/HLSL语义

读者在平时的Shader学习中可能经常看到，在顶点着色器和片元着色器的输入输出变量后还有一个冒号以及一个全部大写的名称，例如在5.2节看到的SV\_POSITION、POSITION、COLOR0。这些大写的名字是什么意思呢？它们有什么用呢？

### 5.4.1 什么是语义

实际上，这些是Cg/HLSL提供的**语义（semantics）**。如果读者从前接触过Cg/HLSL编程的话，可能对这些语义很熟悉。读者可以在微软的关于DirectX的文档（<https://msdn.microsoft.com/en->

us/library/windows/desktop/bb509647(v=vs.85).aspx#VS) 中找到关于语义的详细说明页面。根据文档我们可以知道，语义实际上就是一个赋给Shader输入和输出的字符串，这个字符串表达了这个参数的含义。通俗地讲，这些语义可以让Shader知道从哪里读取数据，并把数据输出到哪里，它们在Cg/HLSL的Shader流水线中是不可或缺的。需要注意的是，Unity并没有支持所有的语义。

通常情况下，这些输入输出变量并不需要有特别的意义，也就是说，我们可以自行决定这些变量的用途。例如在上面的代码中，顶点着色器的输出结构体中我们用COLOR0语义去描述color变量。color变量本身存储了什么，Shader流水线并不关心。

而Unity为了方便对模型数据的传输，对一些语义进行了特别的含义规定。例如，在顶点着色器的输入结构体a2v用TEXCOORD0来描述texcoord，Unity会识别TEXCOORD0语义，以把模型的第一组纹理坐标填充到texcoord中。需要注意的是，即便语义的名称一样，如果出现的位置不同，含义也不同。例如，TEXCOORD0既可以用于描述顶点着色器的输入结构体a2v，也可用于描述输出结构体v2f。但在输入结构体a2v中，TEXCOORD0有特别的含义，即把模型的第一组纹理坐标存储在该变量中，而在输出结构体v2f中，TEXCOORD0修饰的变量含义就可以由我们来决定。

在DirectX 10以后，有了一种新的语义类型，就是**系统数值语义**（**system-value semantics**）。这类语义是以SV开头的，SV代表的含义就是**系统数值**（**system-value**）。这些语义在渲染流水线中有特殊的含义。例如在上面的代码中，我们使用SV\_POSITION语义去修饰顶点着

色器的输出变量pos，那么就表示pos包含了可用于光栅化的变换后的顶点坐标（即齐次裁剪空间中的坐标）。用这些语义描述的变量是不可以随便赋值的，因为流水线需要使用它们来完成特定的目的，例如渲染引擎会把用SV\_POSITION修饰的变量经过光栅化后显示在屏幕上。读者有时可能会看到同一个变量在不同的Shader里面使用了不同的语义修饰。例如，一些Shader会使用POSITION而非SV\_POSITION来修饰顶点着色器的输出。SV\_POSITION是DirectX 10中引入的系统数值语义，在绝大多数平台上，它和POSITION语义是等价的，但在某些平台（例如索尼 PS4）上必须使用SV\_POSITION来修饰顶点着色器的输出，否则无法让Shader正常工作。同样的例子还有COLOR和SV\_Target。因此，为了让我们的Shader有更好的跨平台性，对于这些有特殊含义的变量我们最好使用以SV开头的语义进行修饰。我们在5.6节中会总结更多这种因为平台差异而造成的问题。

### 5.4.2 Unity支持的语义

表5.5总结了从应用阶段传递模型数据给顶点着色器时Unity使用的常用语义。这些语义虽然没有使用SV开头，但Unity内部赋予了它们特殊的含义。

表5.5 从应用阶段传递模型数据给顶点着色器时Unity支持的常用语义

语 义	描 述
POSITION	模型空间中的顶点位置，通常是float4类型
NORMAL	顶点法线，通常是float3类型

语 义	描 述
TANGENT	顶点切线，通常是float4类型
TEXCOORD $n$ ，如 TEXCOORD0、 TEXCOORD1	该顶点的纹理坐标，TEXCOORD0表示第一组纹理坐标，依此类推。通常是float2或float4类型
COLOR	顶点颜色，通常是fixed4或float4类型

其中TEXCOORD $n$ 中 $n$ 的数目是和Shader Model有关的，例如一般在Shader Model 2（即Unity默认编译到的Shader Model版本）和Shader Model 3中， $n$ 等于8，而在Shader Model 4和Shader Model 5中， $n$ 等于16。通常情况下，一个模型的纹理坐标组数一般不超过2，即我们往往只使用TEXCOORD0和TEXCOORD1。在Unity内置的数据结构体appdata\_full中，它最多使用了6个坐标纹理组。

表5.6总结了从顶点着色器阶段到片元着色器阶段Unity支持的常用语义。

表5.6 从顶点着色器传递数据给片元着色器时Unity使用的常用语义

语 义	描 述
-----	-----

语 义	描 述
SV_POSITION	裁剪空间中的顶点坐标，结构体中必须包含一个用该语义修饰的变量。等同于DirectX 9中的POSITION，但最好使用SV_POSITION
COLOR0	通常用于输出第一组顶点颜色，但不是必需的
COLOR1	通常用于输出第二组顶点颜色，但不是必需的
TEXCOORD0 ～ TEXCOORD7	通常用于输出纹理坐标，但不是必需的

上面的语义中，除了`SV_POSITION`是有特别含义外，其他语义对变量的含义没有明确要求，也就是说，我们可以存储任意值到这些语义描述变量中。通常，如果我们需要把一些自定义的数据从顶点着色器传递给片元着色器，一般选用`TEXCOORD0`等。

表5.7给出了Unity中支持的片元着色器的输出语义。

表5.7 片元着色器输出时Unity支持的常用语义

语 义	描 述
SV_Target	输出值将会存储到渲染目标（render target）中。等同于DirectX 9中的COLOR语义，但最好使用SV_Target



### 5.4.3 如何定义复杂的变量类型

上面提到的语义绝大部分用于描述标量或矢量类型的变量，例如 `fixed2`、`float`、`float4`、`fixed4`等。下面的代码给出了一个使用语义来修饰不同类型变量的例子：

```
struct v2f {  
    float4 pos : SV_POSITION;  
    fixed3 color0 : COLOR0;  
    fixed4 color1 : COLOR1;  
    half value0 : TEXCOORD0;  
    float2 value1 : TEXCOORD1;  
};
```

关于何时使用哪种变量类型，我们会在5.7.1节给出一些建议。但需要注意的是，一个语义可以使用的寄存器只能处理4个浮点值

（`float`）。因此，如果我们想要定义矩阵类型，如`float3×4`、`float4×4`等变量就需要使用更多的空间。一种方法是，把这些变量拆分成多个变量，例如对于`float4×4`的矩阵类型，我们可以拆分成4个`float4`类型的变量，每个变量存储了矩阵中的一行数据。

## 5.5 程序员的烦恼：Debug

有这样一个笑话，据说只有程序员才能看懂：

> 问：程序员最讨厌康熙的哪个儿子？

> 答：胤禩。因为他是八阿哥（谐音：`bug`）。

**调试（debug）**，大概是所有程序员的噩梦。而不幸的是，对一个Shader进行调试更是噩梦中的噩梦。这也是造成Shader难写的原因之一

——如果发现得到的效果不对，我们可能要花非常多的时间来找到问题所在。造成这种现状的原因就是在Shader中可以选择的调试方法非常有限，甚至连简单的输出都不行。

本节旨在给出Unity中对Unity Shader的调试方法，这主要包含了两种方法。

### 5.5.1 使用假彩色图像

**假彩色图像 (false-color image)** 指的是用假彩色技术生成的一种图像。与假彩色图像对应的是照片这种**真彩色图像 (true-color image)**。一张假彩色图像可以用于可视化一些数据，那么如何用它来对Shader进行调试呢？

主要思想是，我们可以把需要调试的变量映射到 $[0, 1]$ 之间，把它们作为颜色输出到屏幕上，然后通过屏幕上显示的像素颜色来判断这个值是否正确。读者心里可能已经在咆哮：“什么？！这方法也太原始了吧！”没错，这种方法得到的调试信息很模糊，能够得到的信息很有限，但在很长一段时间内，这种方法的确是唯一的可选方法。

需要注意的是，由于颜色的分量范围在 $[0, 1]$ ，因此我们需要小心处理需要调试的变量的范围。如果我们已知它的值域范围，可以先把它映射到 $[0, 1]$ 之间再进行输出。如果你不知道一个变量的范围（这往往说明你对这个Shader中的运算并不了解），我们就只能不停地实验。一个提示是，颜色分量中任何大于1的数值将会被设置为1，而任何小于0的数值会被设置为0。因此，我们可以尝试使用不同的映射，直到发现颜色发生了变化（这意味着得到了 $0\sim 1$ 的值）。

如果我们要调试的数据是一个一维数据，那么可以选择一个单独的颜色分量（如R分量）进行输出，而把其他颜色分量置为0。如果是多维数据，可以选择对它的每一个分量单独调试，或者选择多个颜色分量进行输出。

作为实例，下面我们会使用假彩色图像的方式来可视化一些模型数据，如法线、切线、纹理坐标、顶点颜色，以及它们之间的运算结果等。我们使用的代码如下：

```
Shader "Unity Shaders Book/Chapter 5/False Color" {
    SubShader {
        Pass {
            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            struct v2f {
                float4 pos : SV_POSITION;
                fixed4 color : COLOR0;
            };

            v2f vert(appdata_full v) {
                v2f o;
                o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

                // 可视化法线方向
                o.color = fixed4(v.normal * 0.5 + fixed3(0.5, 0.5,
0.5), 1.0);

                // 可视化切线方向
                o.color = fixed4(v.tangent.xyz * 0.5 + fixed3(0.5,
0.5, 0.5), 1.0);

                // 可视化副切线方向
                fixed3 binormal = cross(v.normal, v.tangent.xyz) *
v.tangent.w;
                o.color = fixed4(binormal * 0.5 + fixed3(0.5, 0.5,
0.5), 1.0);

                // 可视化第一组纹理坐标
```

```

        o.color = fixed4(v.texcoord.xy, 0.0, 1.0);

        // 可视化第二组纹理坐标
        o.color = fixed4(v.texcoord1.xy, 0.0, 1.0);

        // 可视化第一组纹理坐标的小数部分
        o.color = frac(v.texcoord);
        if (any(saturate(v.texcoord) - v.texcoord)) {
            o.color.b = 0.5;
        }
        o.color.a = 1.0;

        // 可视化第二组纹理坐标的小数部分
        o.color = frac(v.texcoord1);
        if (any(saturate(v.texcoord1) - v.texcoord1)) {
            o.color.b = 0.5;
        }
        o.color.a = 1.0;

        // 可视化顶点颜色
        //o.color = v.color;

        return o;
    }

    fixed4 frag(v2f i) : SV_Target {
        return i.color;
    }

    ENDCG
}
}
}

```

在上面的代码中，我们使用了Unity内置的一个结构体——*appdata\_full*。我们在5.3节讲过该结构体的构成。我们可以在UnityCG.cginc里找到它的定义：

```

struct appdata_full {
    float4 vertex : POSITION;
    float4 tangent : TANGENT;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
    float4 texcoord1 : TEXCOORD1;
    float4 texcoord2 : TEXCOORD2;
    float4 texcoord3 : TEXCOORD3;
#ifdef SHADER_API_XBOX360

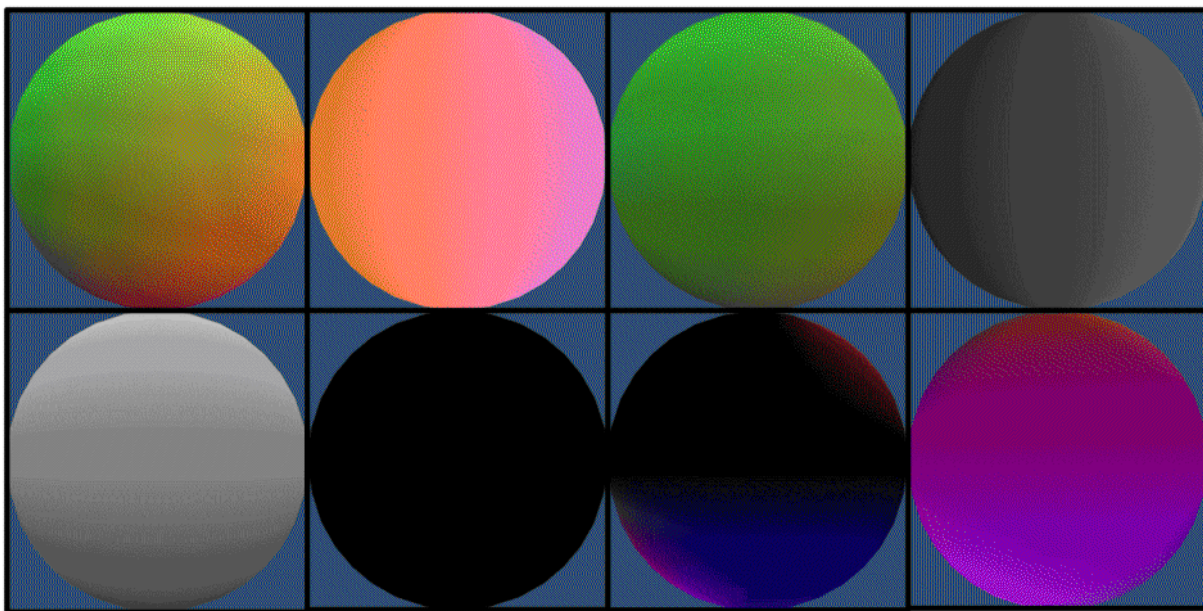
```

```
    half4 texcoord4 : TEXCOORD4;  
    half4 texcoord5 : TEXCOORD5;  
#endif  
    fixed4 color : COLOR;  
};
```

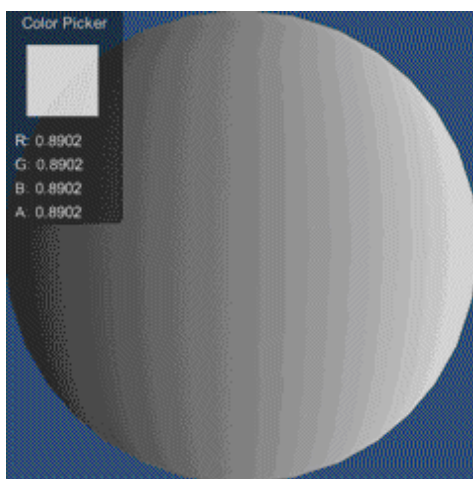
可以看出， `appdata_full` 几乎包含了所有的模型数据。

我们把计算得到的假彩色存储到了顶点着色器的输出结构体——`v2f`中的`color`变量里，并且在片元着色器中输出了这个颜色。读者可以对其中的代码添加或取消注释，观察不同运算和数据得到的效果。图5.4给出了这些代码得到的显示效果。读者可以先自己想一想代码和这些效果之间的对应关系，然后再在Unity中进行验证。

为了可以得到某点的颜色值，我们可以使用类似颜色拾取器的脚本得到屏幕上某点的RGBA值，从而推断出该点的调试信息。在本书的附带工程中，读者可以找到这样一个简单的实例脚本：*Assets -> Scripts -> Chapter5 -> ColorPicker.cs*。把该脚本拖曳到一个摄像机上，单击运行后，可以用鼠标单击屏幕，以得到该点的颜色值，如图5.5所示。



▲ 图5.4 用假彩色对Unity Shader进行调试



▲ 图5.5 使用颜色拾取器来查看调试信息

## 5.5.2 利用神器：Visual Studio

本节是Windows用户的福音，Mac用户的噩耗。Visual Studio作为Windows系统下的开发利器，在Visual Studio 2012版本中也提供了对Unity Shader的调试功能——**Graphics Debugger**。

通过Graphics Debugger，我们不仅可以查看每个像素的最终颜色、位置等信息，还可以对顶点着色器和片元着色器进行单步调试。具体的安装和使用方法可以参见Unity官网文档中**使用Visual Studio对DirectX 11的Shader进行调试**一文（<http://docs.unity3d.com/Manual/SL-Debugging-D3D11ShadersWithVS.html>）。

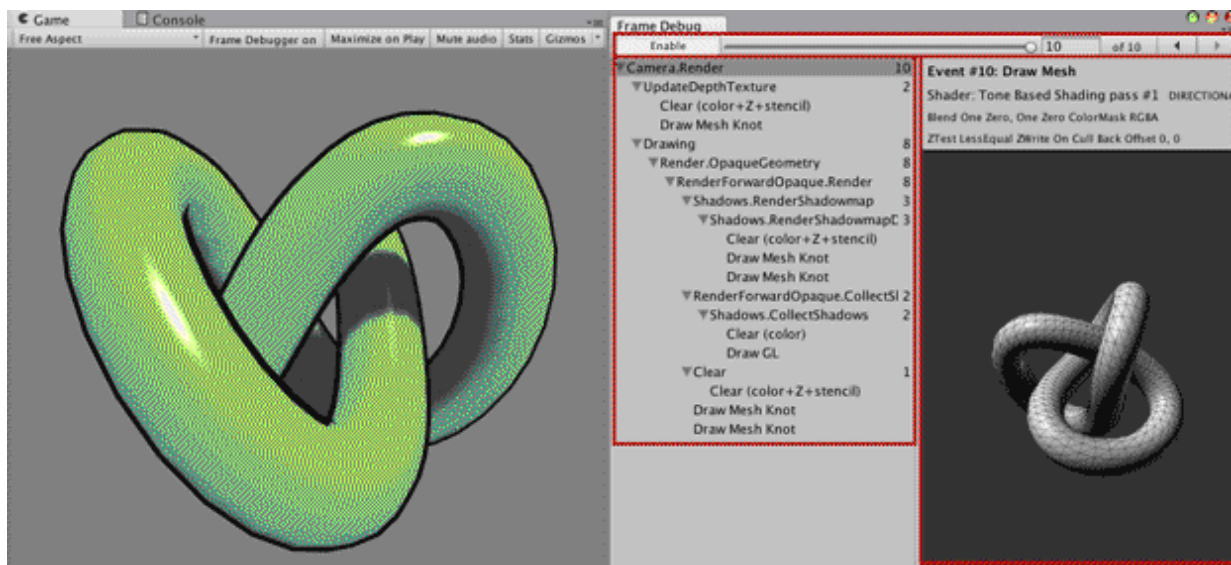
当然，本方法也有一些限制。例如，我们需要保证Unity运行在DirectX 11平台上，而且Graphics Debugger本身存在一些bug。但这无法阻止我们对它的喜爱之情！而Mac用户可能就只能无奈地眼馋了。

### 5.5.3 最新利器：帧调试器

尽管Mac用户无法体验Visual Studio的强大功能，但幸运的是，Unity 5除了带来全新的UI系统外，还给我们带来了一个新的针对渲染的调试器——**帧调试器（Frame Debugger）**。与其他调试工具的复杂性相比，Unity原生的帧调试器非常简单快捷。我们可以使用它来看到游戏图像的某一帧是如何一步步渲染出来的。

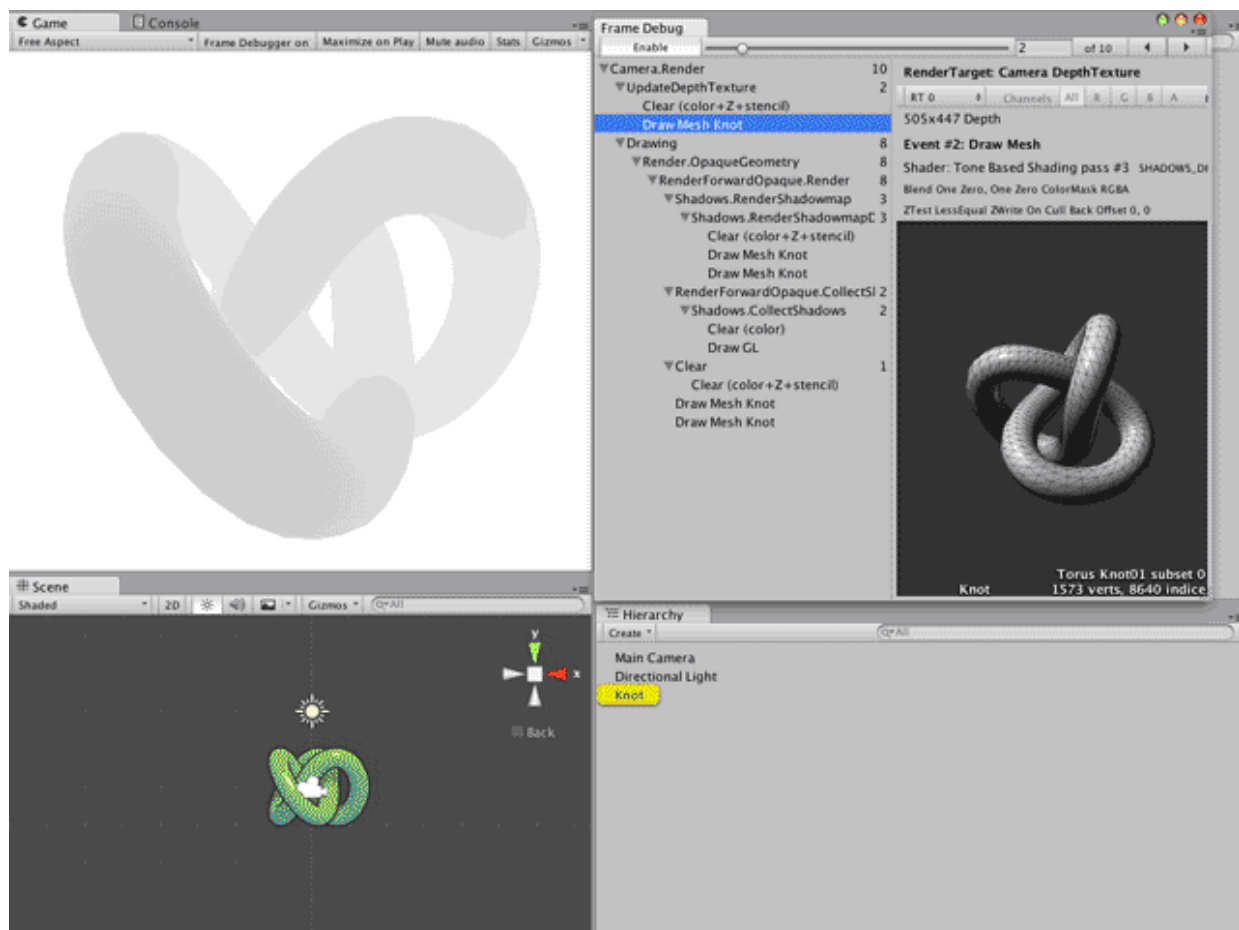
要使用帧调试器，我们首先需要在Window -> *Frame Debugger*中打开帧调试器窗口，如图5.6所示。





▲图5.6 帧调试器

帧调试器可以用于查看渲染该帧时进行的各种**渲染事件**（**event**），这些事件包含了**Draw Call**序列，也包括了类似清空帧缓存等操作。帧调试器窗口大致可分为3个部分：最上面的区域可以开启/关闭（单击**Enable**按钮）帧调试功能，当开启了帧调试时，通过移动窗口最上方的滑动条（或单击前进和后退按钮），我们可以重放这些渲染事件；左侧的区域显示了所有事件的树状图，在这个树状图中，每个叶子节点就是一个事件，而每个父节点的右侧显示了该节点下的事件数目。我们可以从事件的名字了解这个事件的操作，例如以**Draw**开头的事件通常就是一个**Draw Call**；当单击了某个事件时，在右侧的窗口中就会显示出该事件的细节，例如几何图形的细节以及使用了哪个**Shader**等。同时在**Game**视图中我们也可以看到它的效果。如果该事件是一个**Draw Call**并且对应了场景中的一个**GameObject**，那么这个**GameObject**也会在**Hierarchy**视图中被高亮显示出来，图5.7显示了单击渲染某个对象的深度图事件的结果。



▲ 图5.7 单击Knot的深度图渲染事件，在Game视图会显示该事件的效果，在Hierarchy视图中会高亮显示Knot对象，在帧调试器的右侧窗口会显示出该事件的细节

如果被选中的Draw Call是对一个渲染纹理（RenderTexture）的渲染操作，那么这个渲染纹理就会显示在Game视图中。而且，此时右侧面板上方的工具栏中也会出现更多的选项，例如在Game视图中单独显示R、G、B和A通道。

Unity 5提供的帧调试器实际上并没有实现一个真正的帧拾取（frame capture）的功能，而是仅仅使用**停止渲染**的方法来查看渲染事件的结果。例如，如果我们想要查看第4个Draw Call的结果，那么帧调试器就会在第4个Draw Call调用完毕后停止渲染。这种方法虽然简单，

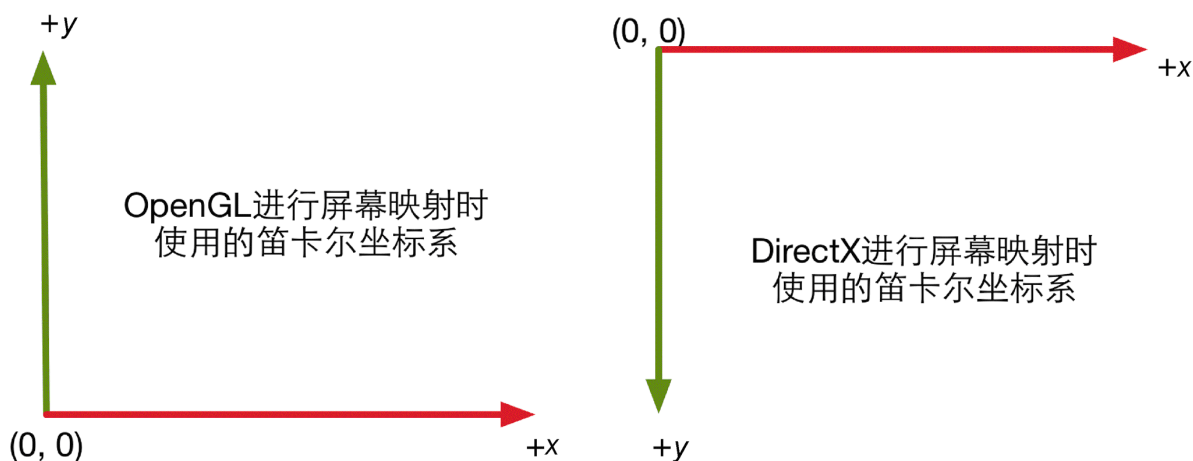
但得到的信息也很有限。如果读者想要获取更多的信息，还是需要使用外部工具，例如5.5.2节中的Visual Studio插件，或者Intel GPA、RenderDoc、NVIDIA NSight、AMD GPU PerfStudio等工具。

## 5.6 小心：渲染平台的差异

Unity的优点之一是其强大的跨平台性——写一份代码可以运行在很多平台上。绝大多数情况下，Unity为我们隐藏了这些细节，但有些时候我们需要自己处理它们。本节给出了一些常见的因为平台不同而造成的差异。

### 5.6.1 渲染纹理的坐标差异

在2.3.4节和4.2.2节中，我们都提到过OpenGL和DirectX的屏幕空间坐标的差异。在水平方向上，两者的数值变化方向是相同的，但在竖直方向上，两者是相反的。在OpenGL（OpenGL ES也是）中， $(0, 0)$ 点对应了屏幕的左下角，而在DirectX（Metal也是）中， $(0, 0)$ 点对应了左上角。图5.8可以帮助读者回忆它们之间的这种不同。



▲ 图5.8 OpenGL和DirectX使用了不同的屏幕空间坐标

需要注意的是，我们不仅可以把渲染结果输出到屏幕上，还可以输出到不同的渲染目标（**Render Target**）中。这时，我们需要使用渲染纹理（**Render Texture**）来保存这些渲染结果。我们将在第12章中学习如何实现这样的目的。

大多数情况下，这样的差异并不会对我们造成任何影响。但当我们使用渲染到纹理技术，把屏幕图像渲染到一张渲染纹理中时，如果不采取任何措施的话，就会出现纹理翻转的情况。幸运的是，Unity在背后为我们处理了这种翻转问题——当在DirectX平台上使用渲染到纹理技术时，Unity会为我们翻转屏幕图像纹理，以便在不同平台上达到一致性。

在一种特殊情况下Unity不会为我们进行这个翻转操作，这种情况就是我们开启了抗锯齿（在Edit -> Project Settings -> Quality -> Anti Aliasing中开启）并在此时使用了渲染到纹理技术。在这种情况下，Unity首先渲染得到屏幕图像，再由硬件进行抗锯齿处理后，得到一张渲染纹理来供我们进行后续处理。此时，在DirectX平台下，我们得到的输入屏幕图像并不会被Unity翻转，也就是说，此时对屏幕图像的采样坐标是需要符合DirectX平台规定的。如果我们的屏幕特效只需要处理一张渲染图像，我们仍然不需要在意纹理的翻转问题，这是因为在我们调用Graphics.Blit函数时，Unity已经为我们对屏幕图像的采样坐标进行了处理，我们只需要按正常的采样过程处理屏幕图像即可。但如果我们需要同时处理多张渲染图像（前提是开启了抗锯齿），例如需要同时处理屏幕图像和法线纹理，这些图像在垂直方向的朝向就可能是不同的（只有在DirectX这样的平台上才有这样的问题）。这种时

候，我们就需要自己在顶点着色器中翻转某些渲染纹理（例如深度纹理或其他由脚本传递过来的纹理）的纵坐标，使之都符合DirectX平台的规则。例如：

```
#if UNITY_UV_STARTS_AT_TOP
if (_MainTex_TexelSize.y < 0)
    uv.y = 1-uv.y;
#endif
```

其中，UNITY\_UV\_STARTS\_AT\_TOP用于判断当前平台是否是DirectX类型的平台，而当在这样的平台下开启了抗锯齿后，主纹理的纹素大小在竖直方向上会变成负值，以方便我们对主纹理进行正确的采样。因此，我们可以通过判断\_MainTex\_TexelSize.y是否小于0来检验是否开启了抗锯齿。如果是，我们就需要对除主纹理外的其他纹理的采样坐标进行竖直方向上的翻转。我们会在第13章中再次看到上面的代码。

在本书资源的项目中，我们开启了抗锯齿选项。在第12章中，我们将学习一些基本的屏幕后处理效果。这些效果大多使用了单张屏幕图像进行处理，因此我们不需要考虑平台差异化的问题，因为Unity已经在背后为我们处理过了。但在12.5节中，我们需要在一个Pass中同时处理屏幕图像和提取得到的亮部图像来实现Bloom效果。由于需要同时处理多张纹理，因此在DirectX这样的平台下如果开启了抗锯齿，主纹理和亮部纹理在竖直方向上的朝向就是不同的，我们就需要对亮部纹理的采样坐标进行翻转。在第13章中，我们需要同时处理屏幕图像和深度/法线纹理来实现一些特殊的屏幕效果，在这些处理过程中，我们也需要进行一些平台差异化处理。在15.3节中，尽管我们也在一个Pass中同时处理了屏幕图像、深度纹理和一张噪声纹理，但我们只对深度

纹理的采样坐标进行了平台差异化处理，而没有对噪声纹理进行处理。这是因为，类似噪声纹理的装饰性纹理，它们在竖直方向上的朝向并不是很重要，即便翻转了效果往往也是正确的，因此我们可以不对这些纹理进行平台差异化处理。

## 5.6.2 Shader的语法差异

读者在Windows平台下编译某些在Mac平台下工作良好的Shader时，可能会看到类似下面的报错信息：

```
incorrect number of arguments to numeric-type constructor  
(compiling for d3d11)
```

或者

```
output parameter 'o' not completely initialized (compiling for  
d3d11)
```

上面的报错都是因为DirectX 9/11对Shader的语义更加严格造成的。例如，造成第一个报错信息的原因是，Shader中可能存在下面这样的代码：

```
// v是float4类型，但在它的构造器中我们仅提供了一个参数  
float4 v = float4(0.0);
```

在OpenGL平台上，上面的代码是合法的，它将得到一个4个分量都是0.0的float4类型的变量。但在DirectX 11平台上，我们必须提供和变量类型相匹配的参数数目。也就是说，我们应该写成：

```
float4 v = float4(0.0, 0.0, 0.0, 0.0);
```

而对于第二个报错信息，往往是出现在表面着色器中。表面着色器的顶点函数（注意，不是顶点着色器）有一个使用了`out`修饰符的参数。如果出现这样的报错信息，可能是因为在顶点函数中没有对这个参数的所有成员变量都进行初始化。我们应该使用类似下面的代码来对这些参数进行初始化：

```
void vert (inout appdata_full v, out Input o) {  
    // 使用Unity内置的UNITY_INITIALIZE_OUTPUT宏对输出结构体o进行初始化  
    UNITY_INITIALIZE_OUTPUT(Input,o);  
    // ...  
}
```

除了上述两点语法不同外，DirectX 9 / 11也不支持在顶点着色器中使用`tex2D`函数。`tex2D`是一个对纹理进行采样的函数，我们在后面的章节中将会具体讲到。之所以DirectX 9 / 11不支持顶点阶段中的`tex2D`运算，是因为在顶点着色器阶段Shader无法得到UV偏导，而`tex2D`函数需要这样的偏导信息（这和纹理采样时使用的数学运算有关）。如果我们的确需要在顶点着色器中访问纹理，需要使用`tex2Dlod`函数来替代，如：

```
tex2Dlod(tex, float4(uv, 0, 0)).
```

而且我们还需要添加`#pragma target 3.0`，因为`tex2Dlod`是Shader Model 3.0中的特性。

### 5.6.3 Shader的语义差异

我们在5.4节讲到了Shader中的语义是什么，其中我们讲到了一些语义在某些平台下是等价的，例如`SV_POSITION`和`POSITION`。但在另一些平台上，这些语义是不等价的。为了让Shader能够在所有平台上正



常工作，我们应该尽可能使用下面的语义来描述Shader的输入输出变量。

- 使用`SV_POSITION`来描述顶点着色器输出的顶点位置。一些Shader使用了`POSITION`语义，但这些Shader无法在索尼PS4平台上或使用了细分着色器的情况下正常工作。
- 使用`SV_Target`来描述片元着色器的输出颜色。一些Shader使用了`COLOR`或者`COLOR0`语义，同样的，这些Shader无法在索尼PS4上正常工作。

#### 5.6.4 其他平台差异

本书只给出了一些最常见的平台差异造成的问题，还有一些差异不再列举。如果读者发现一些Shader在平台A下工作良好，而在平台B下出现了问题，可以去Unity官方文档（<http://docs.unity3d.com/Manual/SL-PlatformDifferences.html>）中寻找更多的资料。

### 5.7 Shader整洁之道

在本章的最后，我们给出一些关于如何规范Shader代码的建议。当然，这些建议并不是绝对正确的，读者可以根据实际情况做出权衡。写出规范的代码不仅是让代码变得漂亮易懂而已，更重要的是，养成这些习惯有助于我们写出高效的代码。

#### 5.7.1 float、half还是fixed

在本书中，我们使用Cg/HLSL来编写Unity Shader中的代码。而在Cg/HLSL中，有3种精度的数值类型：`float`，`half`和`fixed`。这些精度将

决定计算结果的数值范围。表5.8给出了这3种精度在通常情况下的数值范围。

表5.8 Cg/HLSL中3种精度的数值类型

类 型	精    度
float	最高精度的浮点值。通常使用32位来存储
half	中等精度的浮点值。通常使用16位来存储，精度范围是-60 000~+60 000
fixed	最低精度的浮点值。通常使用11位来存储，精度范围是-2.0~+2.0

上面的精度范围并不是绝对正确的，尤其是在不同平台和GPU上，它们实际的精度可能和上面给出的范围不一致。通常来讲。

- 大多数现代的桌面GPU会把所有计算都按最高的浮点精度进行计算，也就是说，float、half、fixed在这些平台上实际是等价的。这意味着，我们在PC上很难看出因为half和fixed精度而带来的不同。
- 但在移动平台的GPU上，它们的确会有不同的精度范围，而且不同精度的浮点值的运算速度也会有所差异。因此，我们应该确保在真正的移动平台上验证我们的Shader。
- fixed精度实际上只在一些较旧的移动平台上有用，在大多数现代的GPU上，它们内部把fixed和half当成同等精度来对待。

尽管有上面的不同，但一个基本建议是，尽可能使用精度较低的类型，因为这可以优化Shader的性能，这一点在移动平台上尤其重要。从它们大体的值域范围来看，我们可以使用fixed类型来存储颜色和单位矢量，如果要存储更大范围的数据可以选择half类型，最差情况下再选择使用float。如果我们的目标平台是移动平台，一定要确保在真实的手机上测试我们的Shader，这一点非常重要。关于移动平台的优化技术，读者可以在第16章中找到更多内容。

### 5.7.2 规范语法

在5.6.2节，我们提到DirectX平台对Shader的语义有更加严格的要求。这意味着，如果我们要发布到DirectX平台上就需要使用更严格的语法。例如，使用和变量类型相匹配的参数数目来对变量进行初始化。

### 5.7.3 避免不必要的计算

如果我们毫无节制地在Shader（尤其是片元着色器）中进行了大量计算，那么我们可能很快就会收到Unity的错误提示：

```
temporary register limit of 8 exceeded
```

或

```
Arithmetic instruction limit of 64 exceeded; 65 arithmetic  
instructions needed to compile program
```

出现这些错误信息大多是因为我们在Shader中进行了过多的运算，使得需要的临时寄存器数目或指令数目超过了当前可支持的数目。读

者需要知道，不同的Shader Target、不同的着色器阶段，我们可使用的临时寄存器和指令数目都是不同的。

通常，我们可以通过指定更高等级的Shader Target来消除这些错误。表5.9给出了Unity目前支持的一些Shader Target。

表5.9      Unity支持的Shader Target

指 令	描 述
#pragma target 2.0	默认的Shader Target等级。相当于Direct3D 9上的Shader Model 2.0，不支持对顶点纹理的采样，不支持显式的LOD纹理采样等
#pragma target 3.0	相当于Direct3D 9上的Shader Model 3.0，支持对顶点纹理的采样等
#pragma target 4.0	相当于Direct3D 10上的Shader Model 4.0，支持几何着色器等
#pragma target 5.0	相当于Direct3D 11上的Shader Model 5.0

需要注意的是，由于Unity版本的不同，Unity支持的Shader Target种类也不同，读者可以在官方手册上找到更为详细的介绍。

读者：什么是Shader Model呢？

我们：**Shader Model**是由微软提出的一套规范，通俗地理解就是它们决定了**Shader**中各个特性（**feature**）的能力（**capability**）。这些特性和能力体现在**Shader**能使用的运算指令数目、寄存器个数等各个方面。**Shader Model**等级越高，**Shader**的能力就越大。具体的细节读者可以参见本章的扩展阅读部分。

虽然更高等级的**Shader Target**可以让我们使用更多的临时寄存器和运算指令，但一个更好的方法是尽可能减少**Shader**中的运算，或者通过预计算的方式来提供更多的数据。

#### 5.7.4 慎用分支和循环语句

在我们学习第一门语言的课上，类似分支、循环语句这样的流程控制语句是最基本的语法之一。但在编写**Shader**的时候，我们要对它们格外小心。

在最开始，**GPU**是不支持在顶点着色器和片元着色器中使用流程控制语句的。随着**GPU**的发展，我们现在已经可以使用**if-else**、**for**和**while**这种流程控制指令了。但是，它们在**GPU**上的实现和在**CPU**上有很大的不同。深究这些指令的底层实现不在本书的讨论范围内，读者可以在本章的扩展阅读中找到更多的内容。大体来说，**GPU**使用了不同于**CPU**的技术来实现分支语句，在最坏的情况下，我们花在一个分支语句的时间相当于运行了所有分支语句的时间。因此，我们不鼓励在**Shader**中使用流程控制语句，因为它们会降低**GPU**的并行处理操作（尽管在现代的**GPU**上已经有了改进）。

如果我们在**Shader**中使用了大量的流程控制语句，那么这个**Shader**的性能可能会成倍下降。一个解决方法是，我们应该尽量把计算向流水线上端移动，例如把放在片元着色器中的计算放到顶点着色器中，或者直接在CPU中进行预计算，再把结果传递给**Shader**。当然，有时我们不可避免地要使用分支语句来进行运算，那么一些建议是：

- 分支判断语句中使用的条件变量最好是常数，即在**Shader**运行过程中不会发生变化；
- 每个分支中包含的操作指令数尽可能少；
- 分支的嵌套层数尽可能少。

### 5.7.5 不要除以0

虽然在用类似C#等高级语言进行编程的时候，我们会谨记不要除以0这个基本常识（就算你没这么做，编辑器可能也会报错），但有时在编写**Shader**的时候我们会忽略这个问题。

例如，我们在**Shader**里写下如下代码：

```
fixed4 frag(v2f i) : SV_Target
{
    return fixed4(0.0/0.0,0.0/0.0, 0.0/0.0, 1.0);
}
```

这样代码的结果往往是不可预测的。在某些渲染平台上，上面的代码不会造成**Shader**的崩溃，但即便不会崩溃得到的结果也是不确定的，有些会得到白色（由无限大截取到1.0），有些会得到黑色，但在另一些平台上，我们的**Shader**可能就会直接崩溃。因此，即便在开发游

戏的平台上，我们看到的结果可能是符合预期的，但在目标平台上可能就会出现问题。

一个解决方法是，对那些除数可能为0的情况，强制截取到非0范围。在一些资料中，读者可能也会看到使用if语句来判断除数是否为0的例子。另一个方法是，使用一个很小的浮点值，例如0.000001来保证分母大于0（前提是原始数值是非负数）。

## 5.8 扩展阅读

读者可以在《GPU精粹2》中的**GPU流程控制**一章<sup>[1]</sup>中更加深入地了解为什么流程控制语句在GPU上会影响性能。在5.7.3节我们提到了Shader中临时寄存器数目和运算指令都有限制，实际上Shader Model对顶点着色器和片元着色器中使用的指令数、临时寄存器、常量寄存器、输入/输出寄存器、纹理等数目都进行了规定。读者可以在Wiki的相关资料<sup>[2]</sup>和HLSL的手册<sup>[3]</sup>中找到更多的内容。

[1] Mark Harris, Ian Buck. "GPU Flow-Control Idioms." In GPU Gems 2. 中译本：GPU精粹2：高性能图形芯片和通用计算编程技巧，法尔译，清华大学出版社，2007年。

[2] High-Level Shading Language, Wiki  
([https://en.wikipedia.org/wiki/High-Level\\_Shading\\_Language](https://en.wikipedia.org/wiki/High-Level_Shading_Language))。

[3] Shader Models vs Shader Profiles, HLSL手册  
([https://msdn.microsoft.com/en-us/library/windows/desktop/bb509626\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/bb509626(v=vs.85).aspx))。





## 第6章 Unity中的基础光照

渲染总是围绕着一个基础问题：我们如何决定一个像素的颜色？从宏观上来说，渲染包含了两大部分：决定一个像素的可见性，决定这个像素上的光照计算。而光照模型就是用于决定在一个像素上进行怎样的光照计算。

我们首先会在6.1节介绍在真实世界中，我们是如何看到一个物体的，以此来帮助读者理解光照模型背后的原理。随后在6.2节中，我们将解释什么是标准光照模型，以及如何在Unity Shader中实现标准光照模型。6.3节介绍如何计算光照模型中的环境光和自发光部分。在6.4节和6.5节中，我们将学习两种最基本的光照模型，并比较逐顶点和逐像素光照的区别。最后，在6.6节中介绍如何使用Unity的内置函数来帮助我们实现这些光照模型。

**需要提醒读者注意的是**，本章着重讲述光照模型的原理，因此实现的Shader往往并不能直接应用到实际项目中（直接使用会缺少阴影、光照衰减等效果）。我们会在9.5节给出包含了完整光照模型的可真正使用的Unity Shader。

### 6.1 我们是如何看到这个世界的

我们可能常常会问类似这样的问题：“这个物体是什么颜色的？”如果读者对小学的自然课还有印象的话，可能还会记得这个问题

是没有意义的：当我们在描述“这个物体是红色的”时，实际上是因为这个物体会反射更多的红光波长，而吸收了其他波长。而如果一个物体在我们看来是黑色的，实际上是因为它吸收了绝大部分的波长。这种物理现象就是本节需要探讨的内容。

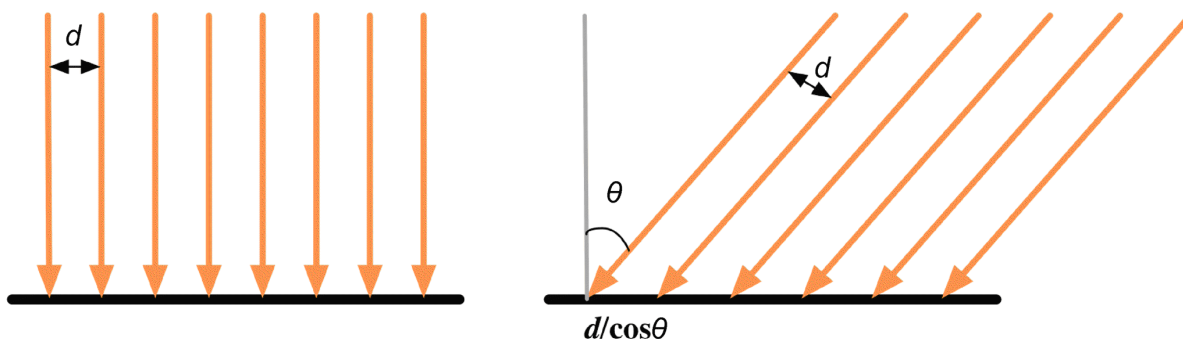
通常来讲，我们要模拟真实的光照环境来生成一张图像，需要考虑3种物理现象。

- 首先，光线从**光源 (light source)** 中被发射出来。
- 然后，光线和场景中的一些物体相交：一些光线被物体吸收了，而另一些光线被散射到其他方向。
- 最后，摄像机吸收了一些光，产生了一张图像。

下面，我们将对每个部分进行更加详细的解释。

### 6.1.1 光源

光不是从石头里蹦出来的，而是由光源发射出来的。在实时渲染中，我们通常把光源当成一个没有体积的点，用 $l$ 来表示它的方向。那么，我们如何测量一个光源发射出了多少光呢？也就是说，我们如何量化光呢？在光学里，我们使用**辐照度 (irradiance)** 来量化光。对于平行光来说，它的辐照度可通过计算在垂直于 $l$ 的单位面积上单位时间内穿过的能量来得到。在计算光照模型时，我们需要知道一个物体表面的辐照度，而物体表面往往是和 $l$ 不垂直的，那么如何计算这样的表面的辐照度呢？我们可以使用光源方向 $l$ 和表面法线 $n$ 之间的夹角的余弦值来得到。需要注意的是，这里默认方向矢量的模都为1。图6.1显示了使用余弦值来计算的原因。



▲图6.1 在左图中，光是垂直照射到物体表面，因此光线之间的垂直距离保持不变；而在右图中，光是斜着照射到物体表面，在物体表面光线之间的距离是 $d/\cos\theta$ ，因此单位面积上接收到的光线数目要少于左图

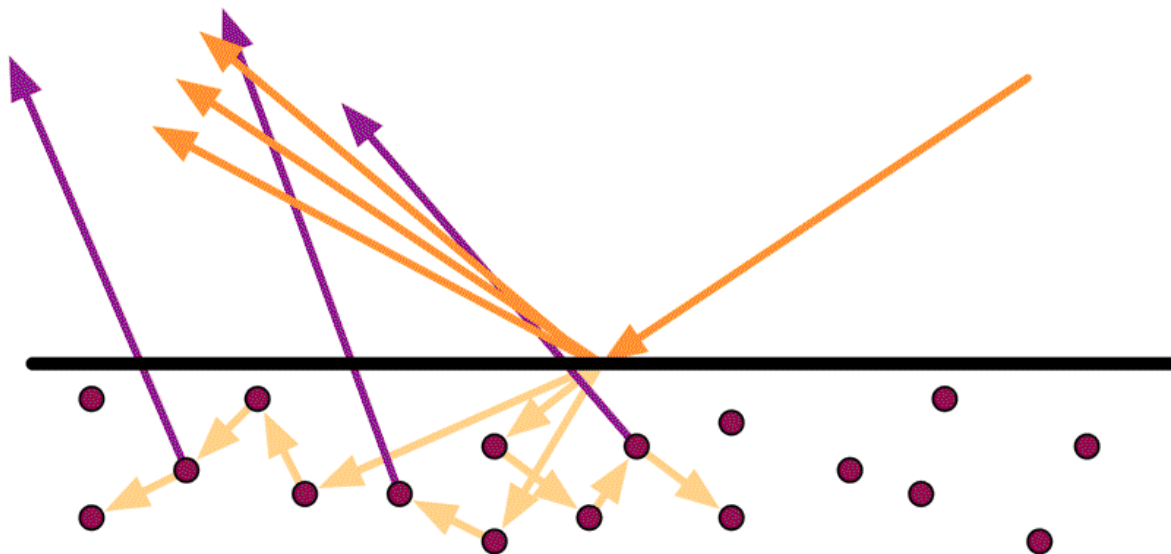
因为辐照度是和照射到物体表面时光线之间的距离 $d/\cos\theta$ 成反比的，因此辐照度就和 $\cos\theta$ 成正比。 $\cos\theta$ 可以使用光源方向 $l$ 和表面法线 $n$ 的点积来得到。这就是使用点积来计算辐照度的由来。

### 6.1.2 吸收和散射

光线由光源发射出来后，就会与一些物体相交。通常，相交的结果有两个：**散射（scattering）**和**吸收（absorption）**。

散射只改变光线的方向，但不改变光线的密度和颜色。而吸收只改变光线的密度和颜色，但不改变光线的方向。光线在物体表面经过散射后，有两种方向：一种将会散射到物体内部，这种现象被称为**折射（refraction）**或**透射（transmission）**；另一种将会散射到外部，这种现象被称为**反射（reflection）**。对于不透明物体，折射进入物体内部的光线还会继续与内部的颗粒进行相交，其中一些光线最后会重新发射出物体表面，而另一些则被物体吸收。那些从物体表面重新发射

出的光线将具有和入射光线不同的方向分布和颜色。图6.2给出了这样的例子。



▲图6.2 散射时，光线会发生折射和反射现象。对于不透明物体，折射的光线会在物体内部继续传播，最终有一部分光线会重新从物体表面被发射出去

为了区分这两种不同的散射方向，我们在光照模型中使用了不同的部分来计算它们：**高光反射（specular）**部分表示物体表面是如何反射光线的，而**漫反射（diffuse）**部分则表示有多少光线会被折射、吸收和散射出表面。根据入射光线的数量和方向，我们可以计算出射光线的数量和方向，我们通常使用**出射度（exitance）**来描述它。辐照度和出射度之间是满足线性关系的，而它们之间的比值就是材质的漫反射和高光反射属性。

在本章中，我们假设漫反射部分是没有方向性的，也就是说，光线在所有方向上是平均分布的。同时，我们也只考虑某一个特定方向上的高光反射。

### 6.1.3 着色

**着色 (shading)** 指的是，根据材质属性（如漫反射属性等）、光源信息（如光源方向、辐照度等），使用一个等式去计算沿某个观察方向的出射度的过程。我们也把这个等式称为**光照模型 (Lighting Model)**。不同的光照模型有不同的目的。例如，一些用于描述粗糙的物体表面，一些用于描述金属表面等。

### 6.1.4 BRDF光照模型

我们已经了解了光线在和物体表面相交时会发生哪些现象。当已知光源位置和方向、视角方向时，我们就需要知道一个表面是如何和光照进行交互的。例如，当光线从某个方向照射到一个表面时，有多少光线被反射？反射的方向有哪些？而**BRDF (Bidirectional Reflectance Distribution Function)** 就是用来回答这些问题的。当给定模型表面上的一个点时，**BRDF**包含了对该点外观的完整的描述。在图形学中，**BRDF**大多使用一个数学公式来表示，并且提供了一些参数来调整材质属性。通俗来讲，当给定入射光线的方向和辐照度后，**BRDF**可以给出在某个出射方向上的光照能量分布。本章涉及的**BRDF**都是对真实场景进行理想化和简化后的模型，也就是说，它们并不能真实地反映物体和光线之间的交互，这些光照模型被称为是经验模型。尽管如此，这些经验模型仍然在实时渲染领域被应用了多年。读者可以从邓恩的著作《3D数学基础：图形与游戏开发》（英文名：《3D Math Primer For Graphics And Game Development》）中提到的一句名言来体会这其中的原因。

计算机图形学的第一定律：如果它看起来是对的，那么它就是对。

然而，有时我们希望可以更加真实地模拟光和物体的交互，这就出现了基于物理的BRDF模型，我们会在第18章基于物理的渲染中看到这些更加复杂的光照模型。

## 6.2 标准光照模型

虽然光照模型有很多种类，但在早期的游戏引擎中往往只使用一个光照模型，这个模型被称为标准光照模型。实际上，在BRDF理论被提出之前，标准光照模型就已经被广泛使用了。

在1973<sup>[1]</sup>年，著名学者裴祥风（Bui Tuong Phong）提出了标准光照模型背后的基本理念。标准光照模型只关心直接光照（direct light），也就是那些直接从光源发射出来照射到物体表面后，经过物体表面的一次反射直接进入摄像机的光线。

它的基本方法是，把进入到摄像机内的光线分为4个部分，每个部分使用一种方法来计算它的贡献度。这4个部分是。

- **自发光（emissive）**部分，本书使用 $c_{emissive}$ 来表示。这个部分用于描述当给定一个方向时，一个表面本身会向该方向发射多少辐射量。需要注意的是，如果没有使用全局光照（global illumination）技术，这些自发光的表面并不会真的照亮周围的物体，而是它本身看起来更亮了而已。



- **高光反射 (specular)** 部分，本书使用  $c_{\text{specular}}$  来表示。这个部分用于描述当光线从光源照射到模型表面时，该表面会在完全镜面反射方向散射多少辐射量。
- **漫反射 (diffuse)** 部分，本书使用  $c_{\text{diffuse}}$  来表示。这个部分用于描述，当光线从光源照射到模型表面时，该表面会向每个方向散射多少辐射量。
- **环境光 (ambient)** 部分，本书使用  $c_{\text{ambient}}$  来表示。它用于描述其他所有的间接光照。

### 6.2.1 环境光

虽然标准光照模型的重点在于描述直接光照，但在真实的世界中，物体也可以被**间接光照 (indirect light)** 所照亮。间接光照指的是，光线通常会在多个物体之间反射，最后进入摄像机，也就是说，在光线进入摄像机之前，经过了不止一次的物体反射。例如，在红地毯上放置一个浅灰色的沙发，那么沙发底部也会有红色，这些红色是由红地毯反射了一部分光线，再反弹到沙发上的。

在标准光照模型中，我们使用了一种被称为环境光的部分来近似模拟间接光照。环境光的计算非常简单，它通常是一个全局变量，即场景中的所有物体都使用这个环境光。下面的等式给出了计算环境光的部分：

$$c_{\text{ambient}} = g_{\text{ambient}}$$

### 6.2.2 自发光

光线也可以直接由光源发射进入摄像机，而不需要经过任何物体的反射。标准光照模型使用自发光来计算这个部分的贡献度。它的计算也很简单，就是直接使用了该材质的自发光颜色：

$$\mathbf{c}_{emissive} = \mathbf{m}_{emissive}$$

通常在实时渲染中，自发光的表面往往并不会照亮周围的表面，也就是说，这个物体并不会被当成一个光源。Unity 5引入的全新的全局光照系统则可以模拟这类自发光物体对周围物体的影响，我们会在第18章中看到。

### 6.2.3 漫反射

漫反射光照是用于对那些被物体表面随机散射到各个方向的辐射度进行建模的。在漫反射中，视角的位置是不重要的，因为反射是完全随机的，因此可以认为在任何反射方向上的分布都是一样的。但是，入射光线的角度很重要。

漫反射光照符合**兰伯特定律 (Lambert's law)**：反射光线的强度与表面法线和光源方向之间夹角的余弦值成正比。因此，漫反射部分的计算如下：

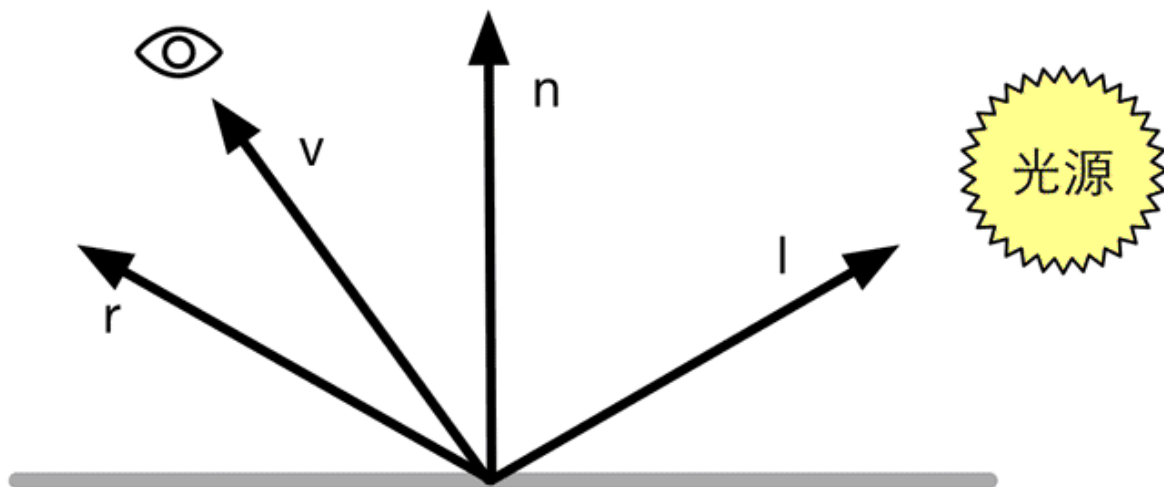
$$\mathbf{c}_{diffuse} = (\mathbf{c}_{light} \cdot \mathbf{m}_{diffuse}) \max(0, \hat{\mathbf{n}} \cdot \hat{\mathbf{l}})$$

其中， $\hat{\mathbf{n}}$ 是表面法线， $\hat{\mathbf{l}}$ 是指向光源的单位矢量， $\mathbf{m}_{diffuse}$ 是材质的漫反射颜色， $\mathbf{c}_{light}$ 是光源颜色。需要注意的是，我们需要防止法线和光源方向点乘的结果为负值，为此，我们使用取最大值的函数来将其截取到0，这可以防止物体被从后面来的光源照亮。

### 6.2.4 高光反射

这里的高光反射是一种经验模型，也就是说，它并不完全符合真实世界中的高光反射现象。它可用于计算那些沿着完全镜面反射方向被反射的光线，这可以让物体看起来是有光泽的，例如金属材质。

计算高光反射需要知道的信息比较多，如表面法线、视角方向、光源方向、反射方向等。在本节中，我们假设这些矢量都是单位矢量。图6.3给出了这些方向矢量。



▲图6.3 使用Phong模型计算高光反射

在这四个矢量中，我们实际上只需要知道其中3个矢量即可，而第四个矢量——反射方向可以通过其他信息计算得到：

$$\hat{r} = 2(\hat{n} \cdot \hat{l})\hat{n} - \hat{l}$$

这样，我们就可以利用Phong模型来计算高光反射的部分：

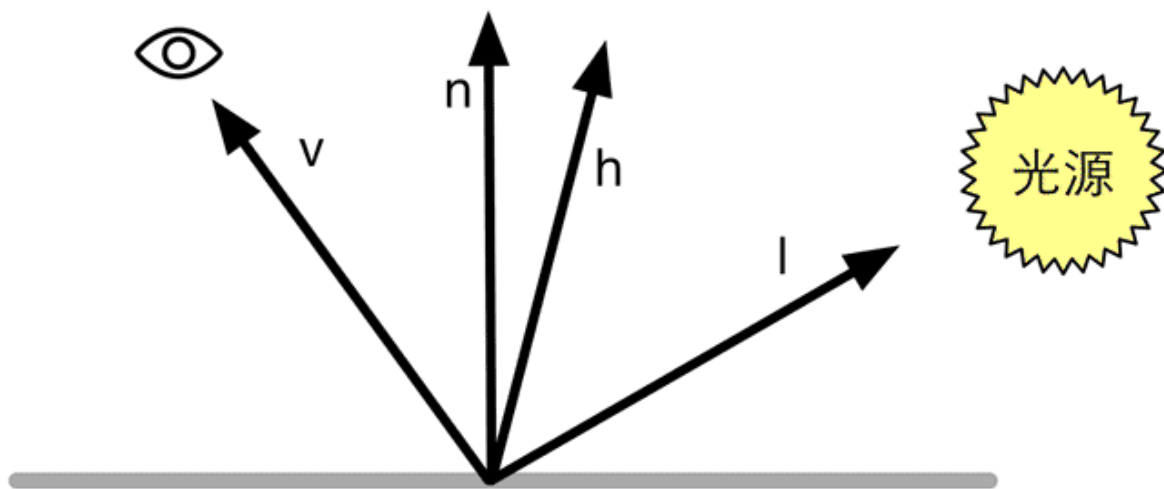
$$c_{specular} = (c_{light} \cdot m_{specular}) \max(0, \hat{v} \cdot \hat{r})^{m_{gloss}}$$

其中， $m_{gloss}$ 是材质的**光泽度（gloss）**，也被称为**反光度（shininess）**。它用于控制高光区域的“亮点”有多宽， $m_{gloss}$ 越大，亮点就越小。 $m_{specular}$ 是材质的高光反射颜色，它用于控制该材质对于高光反射的强度和颜色。 $c_{light}$ 则是光源的颜色和强度。同样，这里也需要防止 $\hat{v} \cdot \hat{r}$ 的结果为负数。

和上述的Phong模型相比，Blinn提出了一个简单的修改方法来得到类似的效果。它的基本思想是，避免计算反射方向 $\hat{r}$ 。为此，Blinn模型引入了一个新的矢量 $\hat{h}$ ，它是通过对 $\hat{v}$ 和 $\hat{l}$ 的取平均后再归一化得到的。即

$$\hat{h} = \frac{\hat{v} + \hat{l}}{|\hat{v} + \hat{l}|}$$

然后，使用 $\hat{n}$ 和 $\hat{h}$ 之间的夹角进行计算，而非 $\hat{v}$ 和 $\hat{r}$ 之间的夹角，如图6.4所示。



▲图6.4 Blinn模型

总结一下，Blinn模型的公式如下：

$$\mathbf{c}_{\text{specular}} = (\mathbf{c}_{\text{light}} \cdot \mathbf{m}_{\text{specular}}) \max(0, \hat{\mathbf{n}} \cdot \hat{\mathbf{h}})^{m_{\text{gloss}}}$$

在硬件实现时，如果摄像机和光源距离模型足够远的话，Blinn模型会快于Phong模型，这是因为，此时可以认为 $\hat{\mathbf{v}}$ 和 $\hat{\mathbf{l}}$ 都是定值，因此 $\hat{\mathbf{h}}$ 将是一个常量。但是，当 $\hat{\mathbf{v}}$ 或者 $\hat{\mathbf{l}}$ 不是定值时，Phong模型可能反而更快一些。需要注意的是，这两种光照模型都是经验模型，也就是说，我们不应该认为Blinn模型是对“正确的”Phong模型的近似。实际上，在一些情况下，Blinn模型更符合实验结果。

### 6.2.5 逐像素还是逐顶点

上面，我们给出了基本光照模型使用的数学公式，那么我们在哪里计算这些光照模型呢？通常来讲，我们有两种选择：在片元着色器中计算，也被称为**逐像素光照（per-pixel lighting）**；在顶点着色器中计算，也被称为**逐顶点光照（per-vertex lighting）**。

在逐像素光照中，我们会以每个像素为基础，得到它的法线（可以是对顶点法线插值得到的，也可以是从法线纹理中采样得到的），然后进行光照模型的计算。这种在面片之间对顶点法线进行插值的技术被称为**Phong着色（Phong shading）**，也被称为Phong插值或法线插值着色技术。这不同于我们之前讲到的Phong光照模型。

与之相对的是逐顶点光照，也被称为**高洛德着色（Gouraud shading）**。在逐顶点光照中，我们在每个顶点上计算光照，然后会在

渲染图元内部进行线性插值，最后输出成像素颜色。由于顶点数目往往远小于像素数目，因此逐顶点光照的计算量往往要小于逐像素光照。但是，由于逐顶点光照依赖于线性插值来得到像素光照，因此，当光照模型中有非线性的计算（例如计算高光反射时）时，逐顶点光照就会出问题。在后面的章节中，我们将会看到这种情况。而且，由于逐顶点光照会在渲染图元内部对顶点颜色进行插值，这会导致渲染图元内部的颜色总是暗于顶点处的最高颜色值，这在某些情况下会产生明显的棱角现象。

### 6.2.6 总结

虽然标准光照模型仅仅是一个经验模型，也就是说，它并不完全符合真实世界中的光照现象。但由于它的易用性、计算速度和得到的效果都比较好，因此仍然被广泛使用。而也是由于它的广泛使用性，这种标准光照模型有很多不同的叫法。例如，一些资料中称它为**Phong光照模型**，因为裴祥风（Bui Tuong Phong）首先提出了使用漫反射和高光反射的和来对反射光照进行建模的基本思想，并且提出了基于经验的计算高光反射的方法（用于计算漫反射光照的兰伯特模型在那时已经被提出了）。而后，由于Blinn的方法简化了计算而且在某些情况下计算更快，我们把这种模型称为**Blinn-Phong光照模型**。

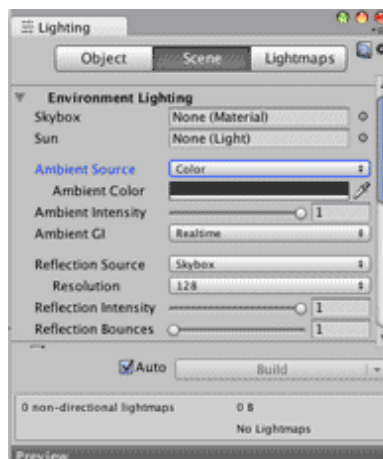
但这种模型有很多局限性。首先，有很多重要的物理现象无法用Blinn-Phong模型表现出来，例如**菲涅耳反射（Fresnel reflection）**。其次，Blinn-Phong模型是**各项同性（isotropic）**的，也就是说，当我们固定视角和光源方向旋转这个表面时，反射不会发生任何改变。但有些表面是具有**各向异性（anisotropic）**反射性质的，例如拉丝金属、毛发

等。在第18章中，我们将学习基于物理的光照模型，这些光照模型更加复杂，同时也可以更加真实地反映光和物体的交互。

## 6.3 Unity中的环境光和自发光

在标准光照模型中，环境光和自发光的计算是最简单的。

在Unity中，场景中的环境光可以在`Window -> Lighting -> Ambient Source/Ambient Color/Ambient Intensity`中控制，如图6.5所示。在Shader中，我们只需要通过Unity的内置变量`UNITY_LIGHTMODEL_AMBIENT`就可以得到环境光的颜色和强度信息。



▲ 图6.5 在Unity的`Window -> Lighting`面板中，我们可以通过`Ambient Source/Ambient Color/Ambient Intensity`来控制场景中的环境光的颜色和强度

而大多数物体是没有自发光特性的，因此在本书绝大部分的Shader中都没有计算自发光部分。如果要计算自发光也非常简单，我们只需要在片元着色器输出最后颜色之前，把材质的自发光颜色添加到输出颜色上即可。



## 6.4 在Unity Shader中实现漫反射光照模型

在了解了上述的理论后，我们现在来看一下如何在Unity中实现这些基本光照模型。首先，我们来实现标准光照模型中的漫反射光照部分。

在6.2.3节中，我们给出了基本光照模型中漫反射部分的计算公式：

$$c_{diffuse} = (c_{light} \cdot m_{diffuse}) \max(0, \hat{n} \cdot \hat{l})$$

从公式可以看出，要计算漫反射需要知道4个参数：入射光线的颜色和强度 $c_{light}$ ，材质的漫反射系数 $m_{diffuse}$ ，表面法线 $\hat{n}$ 以及光源方向 $\hat{l}$ 。

为了防止点积结果为负值，我们需要使用max操作，而Cg提供了这样的函数。在本例中，使用Cg的另一个函数可以达到同样的目的，即saturate函数。

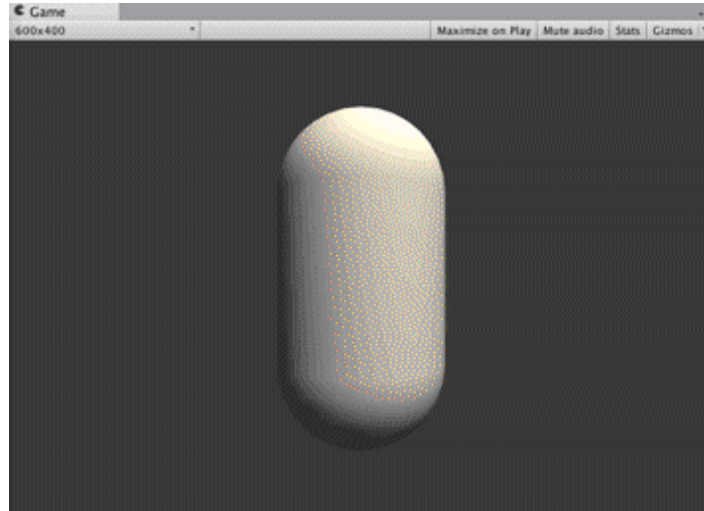
**函数：** saturate(x)

**参数：** x： 为用于操作的标量或矢量，可以是float、float2、float3等类型。

**描述：** 把x截取在[0, 1]范围内，如果x是一个矢量，那么会对它的每一个分量进行这样的操作。

### 6.4.1 实践：逐顶点光照

我们首先来看如何实现一个逐顶点的漫反射光照效果。在学习完本节后，我们会得到类似图6.6中的效果。



▲ 图6.6 逐顶点的漫反射光照效果

为此，我们进行如下准备工作。

(1) 在Unity中新建一个场景。在本书资源中，该场景名为Scene\_6\_4。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window -> Lighting -> Skybox中去掉场景中的天空盒子。

(2) 新建一个材质。在本书资源中，该材质名为DiffuseVertexLevelMat。

(3) 新建一个Unity Shader。在本书资源中，该Shader名为Chapter6-DiffuseVertexLevel。把新的Shader赋给第2步中创建的材质。

(4) 在场景中创建一个胶囊体，并把第2步中的材质赋给该胶囊体。

(5) 保存场景。

下面，我们需要编写自己的Shader来实现一个逐顶点的漫反射效果。打开第3步中创建的Unity Shader，删除所有已有代码，并进行如下修改。

(1) 首先，我们需要为这个Shader起一个名字：

```
Shader "Unity Shaders Book/Chapter 6/Diffuse Vertex-Level" {
```

(2) 为了得到并且控制材质的漫反射颜色，我们首先在Shader的 *Properties* 语义块中声明了一个Color类型的属性，并把它的初始值设为白色：

```
Properties {  
    _Diffuse ("Diffuse", Color) = (1, 1, 1, 1)  
}
```

(3) 然后，我们在 *SubShader* 语义块中定义了一个 *Pass* 语义块。这是因为顶点/片元着色器的代码需要写在 *Pass* 语义块，而非 *SubShader* 语义块中。而且，我们在 *Pass* 的第一行指明了该 *Pass* 的光照模式：

```
SubShader {  
    Pass {  
        Tags { "LightMode"="ForwardBase" }  
    }
```

**LightMode** 标签是 **Pass** 标签中的一种，它用于定义该 **Pass** 在Unity的光照流水线中的角色，在第9章中我们会更加详细地解释它。在这里，

我们只需要知道，只有定义了正确的LightMode，我们才能得到一些Unity的内置光照变量，例如下面要讲到的\_LightColor0。

(4) 然后，我们使用CGPROGRAM和ENDCG来包围Cg代码片，以定义最重要的顶点着色器和片元着色器代码。首先，我们使用#pragma指令来告诉Unity，我们定义的顶点着色器和片元着色器叫什么名字。在本例中，它们的名字分别是vert和frag：

```
CGPROGRAM  
  
#pragma vertex vert  
#pragma fragment frag
```

(5) 为了使用Unity内置的一些变量，如后面要讲到的\_LightColor0，还需要包含进Unity的内置文件Lighting.cginc：

```
#include "Lighting.cginc"
```

(6) 为了在Shader中使用Properties语义块中声明的属性，我们需要定义一个和该属性类型相匹配的变量：

```
fixed4 _Diffuse;
```

通过这样的方式，我们就可以得到漫反射公式中需要的参数之一——材质的漫反射属性。由于颜色属性的范围在0到1之间，因此我们可以使用fixed精度的变量来存储它。

(7) 然后，我们定义了顶点着色器的输入和输出结构体（输出结构体同时也是片元着色器的输入结构体）：

```
struct a2v {  
    float4 vertex : POSITION;
```

```

    float3 normal : NORMAL;
};

struct v2f {
    float4 pos : SV_POSITION;
    fixed3 color : COLOR;
};

```

为了访问顶点的法线，我们需要在a2v中定义一个normal变量，并通过使用NORMAL语义来告诉Unity要把模型顶点的法线信息存储到normal变量中。为了把在顶点着色器中计算得到的光照颜色传递给片元着色器，我们需要在v2f中定义一个color变量，且并不是必须使用COLOR语义，一些资料中会使用TEXCOORD0语义。

(8) 接下来是关键顶点着色器。由于本小节关注如何实现一个逐顶点的漫反射光照，因此漫反射部分的计算都将在顶点着色器中进行：

```

v2f vert(a2v v) {
    v2f o;
    // Transform the vertex from object space to projection space
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    // Get ambient term
    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;

    // Transform the normal from object space to world space
    fixed3 worldNormal = normalize(mul(v.normal,
(float3x3)_World2Object));
    // Get the light direction in world space
    fixed3 worldLight = normalize(_WorldSpaceLightPos0.xyz);
    // Compute diffuse term
    fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb *
    saturate(dot(worldNormal, worldLight));

    o.color = ambient + diffuse;

    return o;
}

```

在第一行，我们首先定义了返回值`o`。我们已经重复过很多次，顶点着色器最基本的任务就是把顶点位置从模型空间转换到裁剪空间中，因此我们需要使用Unity内置的模型世界投影矩阵`UNITY_MATRIX_MVP`来完成这样的坐标变换。接下来，我们通过Unity的内置变量`UNITY_LIGHTMODEL_AMBIENT`得到了环境光部分。

然后，就是真正计算漫反射光照的部分。回忆一下，为了计算漫反射光照我们需要知道4个参数。在前面的步骤中，我们已经知道了材质的漫反射颜色`_Diffuse`以及顶点法线`v.normal`。我们还需要知道光源的颜色和强度信息以及光源方向。Unity提供给我们一个内置变量`_LightColor0`来访问该Pass处理的光源的颜色和强度信息（注意，想要得到正确的值需要定义合适的`LightMode`标签），而光源方向可以由`_WorldSpaceLightPos0`来得到。需要注意的是，这里对光源方向的计算并不具有通用性。在本节中，我们假设场景中只有一个光源且该光源的类型是平行光。但如果场景中有多光源并且类型可能是点光源等其他类型，直接使用`_WorldSpaceLightPos0`就不能得到正确的结果。我们将在6.6节中学习如何使用内置函数来处理更复杂的光源类型。

在计算法线和光源方向之间的点积时，我们需要选择它们所在的坐标系，只有两者处于同一坐标空间下，它们的点积才有意义。在这里，我们选择了世界坐标空间。而由`a2v`得到的顶点法线是位于模型空间下的，因此我们首先需要把法线转换到世界空间中。在4.7节中，我们已经知道可以使用顶点变换矩阵的逆转置矩阵对法线进行相同的变换，因此我们首先得到模型空间到世界空间的变换矩阵的逆矩阵`_World2Object`，然后通过调换它在`mul`函数中的位置，得到和转置矩阵

相同的矩阵乘法。由于法线是一个三维矢量，因此我们只需要截取 `_World2Object` 的前三行前三列即可。

在得到了世界空间中的法线和光源方向后，我们需要对它们进行归一化操作。在得到它们点积的结果后，我们需要防止这个结果为负值。为此，我们使用了 `saturate` 函数。`saturate` 函数是 Cg 提供的一种函数，它的作用是可以把参数截取到 `[0, 1]` 的范围内。最后，再与光源的颜色和强度以及材质的漫反射颜色相乘即可得到最终的漫反射光照部分。

最后，我们对环境光和漫反射光部分相加，得到最终的光照结果。

(9) 由于所有的计算在顶点着色器中都已经完成了，因此片元着色器的代码很简单，我们只需要直接把顶点颜色输出即可：

```
fixed4 frag(v2f i) : SV_Target {  
    return fixed4(i.color, 1.0);  
}
```

(10) 最后，我们需要把这个 Unity Shader 的回调 shader 设置为内置的 `Diffuse`：

```
Fallback "Diffuse"
```

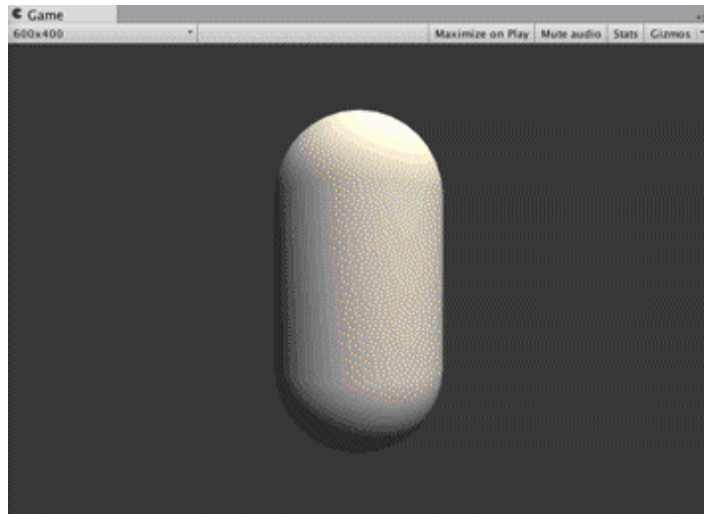
至此，我们已经详细解释了逐顶点的漫反射光照的实现。对于细分程度较高的模型，逐顶点光照已经可以得到比较好的光照效果了。但对于一些细分程度较低的模型，逐顶点光照就会出现一些视觉问



题，例如我们可以在图6.6中看到在胶囊体的背光面与向光面交界处有一些锯齿。为了解决这些问题，我们可以使用逐像素的漫反射光照。

### 6.4.2 实践：逐像素光照

我们只需要对Shader进行一些更改就可以实现逐像素的漫反射效果，如图6.7所示。



▲ 图6.7 逐像素的漫反射光照效果

为此，我们进行如下准备工作。

- (1) 使用6.4.1节中使用的场景。
- (2) 新建一个材质。在本书资源中，该材质名为 `DiffusePixelLevelMat`。
- (3) 新建一个Unity Shader。在本书资源中，该Shader名为 `Chapter6-DiffusePixelLevel`。把新的Shader赋给第2步中创建的材质。

(4) 把第2步中创建的材质赋给胶囊体。

Chapter6-DiffusePixelLevel的代码和6.4.1小节中的非常相似，因此我们首先把6.4.1节中的代码直接粘贴到Chapter6-DiffusePixelLevel中，并进行如下修改。

(1) 修改顶点着色器的输出结构体v2f:

```
struct v2f {  
    float4 pos : SV_POSITION;  
    float3 worldNormal : TEXCOORD0;  
};
```

(2) 顶点着色器不需要计算光照模型，只需要把世界空间下的法线传递给片元着色器即可:

```
v2f vert(a2v v) {  
    v2f o;  
    // Transform the vertex from object space to projection space  
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);  
  
    // Transform the normal from object space to world space  
    o.worldNormal = mul(v.normal, (float3x3)_World2Object);  
  
    return o;  
}
```

(3) 片元着色器需要计算漫反射光照模型:

```
fixed4 frag(v2f i) : SV_Target {  
    // Get ambient term  
    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;  
  
    // Get the normal in world space  
    fixed3 worldNormal = normalize(i.worldNormal);  
    // Get the light direction in world space  
    fixed3 worldLightDir = normalize(_WorldSpaceLightPos0.xyz);  
  
    // Compute diffuse term  
    fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb *  
    ambient + worldNormal * worldLightDir * _Diffuse.rgb *  
    ambient;
```

```
saturate(dot(worldNormal, worldLightDir));  
    fixed3 color = ambient + diffuse;  
    return fixed4(color, 1.0);  
}
```

上面的计算过程和6.4.1节完全相同，这里不再赘述。

逐像素光照可以得到更加平滑的光照效果。但是，即便使用了逐像素漫反射光照，有一个问题仍然存在。在光照无法到达的区域，模型的外观通常是全黑的，没有任何明暗变化，这会使模型的背光区域看起来就像一个平面一样，失去了模型细节表现。实际上我们可以通过添加环境光来得到非全黑的效果，但即便这样仍然无法解决背光面明暗一样的缺点。为此，有一种改善技术被提出来，这就是**半兰伯特（Half Lambert）光照模型**。

### 6.4.3 半兰伯特模型

在6.4.1小节中，我们使用的漫反射光照模型也被称为兰伯特光照模型，因为它符合兰伯特定律——在平面某点漫反射光的光强与该反射点的法向量和入射光角度的余弦值成正比。为了改善6.4.2小节最后提出的问题，Valve公司在开发游戏《半条命》时提出了一种技术，由于该技术是在原兰伯特光照模型的基础上进行了一个简单的修改，因此被称为**半兰伯特光照模型**。

广义的半兰伯特光照模型的公式如下：

$$\mathbf{c}_{diffuse} = (\mathbf{c}_{light} \cdot \mathbf{m}_{diffuse})(\alpha(\hat{\mathbf{n}} \cdot \hat{\mathbf{l}}) + \beta)$$

可以看出，与原兰伯特模型相比，半兰伯特光照模型没有使用 $\max$ 操作来防止 $\hat{n}$ 和 $\hat{l}$ 的点积为负值，而是对其结果进行了一个 $\alpha$ 倍的缩放再加上一个 $\beta$ 大小的偏移。绝大多数情况下， $\alpha$ 和 $\beta$ 的值均为0.5，即公式为：

$$c_{diffuse} = (c_{light} \cdot m_{diffuse})(0.5(\hat{n} \cdot \hat{l}) + 0.5)$$

通过这样的方式，我们可以把 $\hat{n} \cdot \hat{l}$ 的结果范围从 $[-1, 1]$ 映射到 $[0, 1]$ 范围内。也就是说，对于模型的背光面，在原兰伯特光照模型中点积结果将映射到同一个值，即0值处；而在半兰伯特模型中，背光面也可以有明暗变化，不同的点积结果会映射到不同的值上。

需要注意的是，半兰伯特是没有任何物理依据的，它仅仅是一个视觉加强技术。

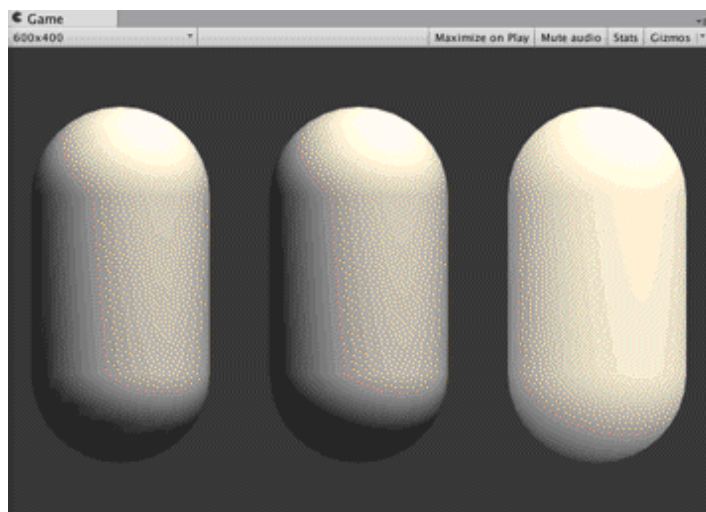
对6.4.2小节中得到的代码做一些修改就可以实现半兰伯特漫反射光照效果。

- (1) 仍然使用6.4.1小节中使用的场景。
- (2) 新建一个材质。在本书资源中，该材质名为HalfLambertMat。
- (3) 新建一个Unity Shader。在本书资源中，该Shader名为Chapter6-HalfLambert。把新的Shader赋给第2步中创建的材质。
- (4) 把第2步中创建的材质赋给胶囊体。

打开Chapter6-HalfLambert，删除已有的Shader代码，把6.4.2小节的Chapter6-DiffusePixelLevel代码粘贴进去，并使用半兰伯特公式修改片元着色器中计算漫反射光照的部分：

```
fixed4 frag(v2f i) : SV_Target {  
    ...  
    // Compute diffuse term  
    fixed halfLambert = dot(worldNormal, worldLightDir) * 0.5 +  
0.5;  
    fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb * halfLambert;  
    fixed3 color = ambient + diffuse;  
    return fixed4(color, 1.0);  
}
```

在上面的代码中，我们使用半兰伯特模型代替了原有的兰伯特模型。图6.8给出了逐顶点漫反射光照、逐像素漫反射光照和半兰伯特光照的对比效果。



▲图6.8 逐顶点漫反射光照、逐像素漫反射光照、半兰伯特光照的对比效果

## 6.5 在Unity Shader中实现高光反射光照模型

在6.2.4节中，我们给出了基本光照模型中高光反射部分的计算公式：

$$\mathbf{c}_{specular} = (\mathbf{c}_{light})\mathbf{m}_{specular})\max(0, \hat{\mathbf{v}} \cdot \hat{\mathbf{r}})^{m_{gloss}}$$

从公式可以看出，要计算高光反射需要知道4个参数：入射光线的颜色和强度 $\mathbf{c}_{light}$ ，材质的高光反射系数 $\mathbf{m}_{specular}$ ，视角方向 $\hat{\mathbf{v}}$ 以及反射方向 $\hat{\mathbf{r}}$ 。其中，反射方向 $\hat{\mathbf{r}}$ 可以由表面法线 $\hat{\mathbf{n}}$ 和光源方向 $\hat{\mathbf{l}}$ 计算而得：

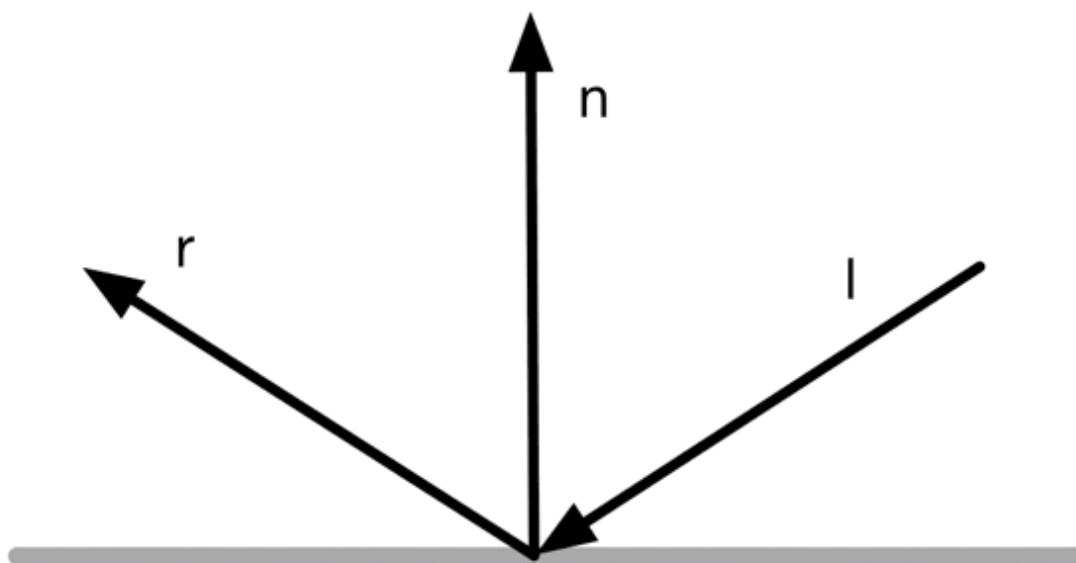
$$\hat{\mathbf{r}} = \hat{\mathbf{l}} - 2(\hat{\mathbf{n}} \cdot \hat{\mathbf{l}})\hat{\mathbf{n}}$$

上述公式很简单，更幸运的是，Cg提供了计算反射方向的函数reflect。

**函数：**reflect(i, n)

**参数：**i，入射方向；n，法线方向。可以是float、float2、float3等类型。

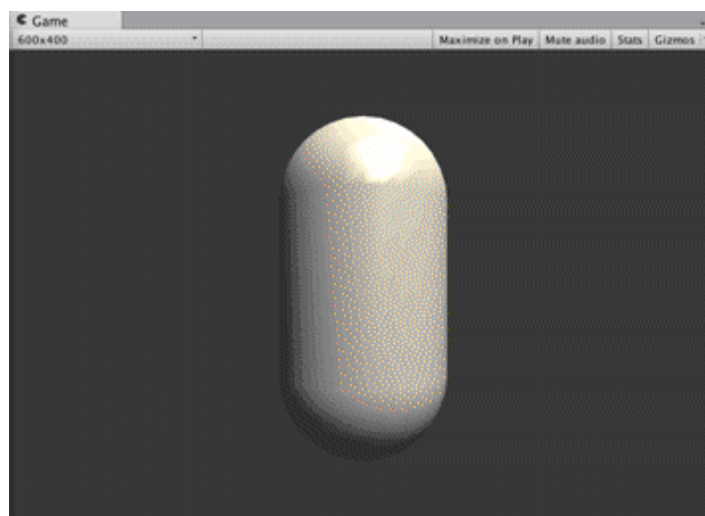
**描述：**当给定入射方向i和法线方向n时，reflect函数可以返回反射方向。图6.9给出了参数和返回值之间的关系。



▲ 图6.9 Cg的reflect函数

### 6.5.1 实践：逐顶点光照

我们首先来看如何实现一个逐顶点的高光反射光照效果。在学习完本节后，我们会得到类似图6.10中的效果。



▲ 图6.10 逐顶点的高光反射光照效果



我们需要进行如下准备工作。

(1) 在Unity中新建一个场景。在本书资源中，该场景名为Scene\_6\_5。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window -> Lighting -> Skybox中去掉场景中的天空盒子。

(2) 新建一个材质。在本书资源中，该材质名为SpecularVertexLevelMat。

(3) 新建一个Unity Shader。在本书资源中，该Shader名为Chapter6-SpecularVertexLevel。把新的Shader赋给第2步中创建的材质。

(4) 在场景中创建一个胶囊体，并把第2步中的材质赋给该胶囊体。

(5) 保存场景。

下面，我们需要编写自己的Shader来实现一个逐顶点的高光反射效果。打开第3步中创建的Chapter6-SpecularVertexLevel，删除所有已有代码，并进行如下修改。

(1) 首先，我们需要为这个Shader起一个名字：

```
Shader "Unity Shaders Book/Chapter 6/Specular Vertex-Level" {
```

(2) 为了在材质面板中能够方便地控制高光反射属性，我们在Shader的*Properties*语义块中声明了三个属性：

```
Properties {  
    _Diffuse ("Diffuse", Color) = (1, 1, 1, 1)  
    _Specular ("Specular", Color) = (1, 1, 1, 1)  
    _Gloss ("Gloss", Range(8.0, 256)) = 20  
}
```

其中，新添加的\_Specular用于控制材质的高光反射颜色，而\_Gloss用于控制高光区域的大小。

(3) 然后，我们在`SubShader`语义块中定义了一个`Pass`语义块。这是因为顶点/片元着色器的代码需要写在`Pass`语义块，而非`SubShader`语义块中。而且，我们在`Pass`的第一行指明了该`Pass`的光照模式：

```
SubShader {  
    Pass {  
        Tags { "LightMode"="ForwardBase" }  
    }  
}
```

`LightMode`标签是`Pass`标签中的一种，它用于定义该`Pass`在Unity的光照流水线中的角色，在第9章中我们会更加详细地解释它。在这里，我们只需要知道，只有定义了正确的`LightMode`，我们才能得到一些Unity的内置光照变量，例如`_LightColor0`。

(4) 然后，我们使用`CGPROGRAM`和`ENDCG`来包围CG代码片，以定义最重要的顶点着色器和片元着色器代码。首先，我们使用`#pragma`指令来告诉Unity，我们定义的顶点着色器和片元着色器叫什么名字。在本例中，它们的名字分别是`vert`和`frag`：

```
CGPROGRAM  
  
#pragma vertex vert  
#pragma fragment frag
```

(5) 为了使用Unity内置的一些变量，如\_LightColor0，还需要包含进Unity的内置文件Lighting.cginc:

```
#include "Lighting.cginc"
```

(6) 为了在Shader中使用*Properties*语义块中声明的属性，我们需要定义和这些属性类型相匹配的变量:

```
fixed4 _Diffuse;  
fixed4 _Specular;  
float _Gloss;
```

由于颜色属性的范围在0到1之间，因此对于\_Diffuse和\_Specular属性我们可以使用fixed精度的变量来存储它。而\_Gloss的范围很大，因此我们使用float精度来存储。

(7) 然后，我们定义了顶点着色器的输入和输出结构体（输出结构体同时也是片元着色器的输入结构体）：

```
struct a2v {  
    float4 vertex : POSITION;  
    float3 normal : NORMAL;  
};  
  
struct v2f {  
    float4 pos : SV_POSITION;  
    fixed3 color : COLOR;  
};
```

(8) 在顶点着色器中，我们计算了包含高光反射的光照模型:

```
v2f vert(a2v v) {  
    v2f o;  
    // Transform the vertex from object space to projection space  
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);  
  
    // Get ambient term
```

```

    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;

    // Transform the normal from object space to world space
    fixed3 worldNormal = normalize(mul(v.normal,
(float3x3)_World2Object));
    // Get the light direction in world space
    fixed3 worldLightDir = normalize(_WorldSpaceLightPos0.xyz);

    // Compute diffuse term
    fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb *
saturate(dot(worldNormal, worldLightDir));

    // Get the reflect direction in world space
    fixed3 reflectDir = normalize(reflect(-worldLightDir,
worldNormal));
    // Get the view direction in world space
    fixed3 viewDir = normalize(_WorldSpaceCameraPos.xyz -
mul(_Object2World, v.vertex).xyz);

    // Compute specular term
    fixed3 specular = _LightColor0.rgb * _Specular.rgb *
pow(saturate(dot(reflectDir, viewDir)), _Gloss);

    o.color = ambient + diffuse + specular;

    return o;
}

```

其中漫反射部分的计算和6.4节中的代码完全一致。对于高光反射部分，我们首先计算了入射光线方向关于表面法线的反射方向 **reflectDir**。由于Cg的reflect函数的入射方向要求是由光源指向交点处的，因此我们需要对**worldLightDir**取反后再传给reflect函数。然后，我们通过**\_WorldSpaceCameraPos**得到了世界空间中的摄像机位置，再把顶点位置从模型空间变换到世界空间下，再通过和**\_WorldSpaceCameraPos**相减即可得到世界空间下的视角方向。

由此，我们已经得到了所有的4个参数，代入公式即可得到高光反射的光照部分。最后，再和环境光、漫反射光相加存储到最后的颜色中。

(9) 片元着色器的代码非常简单，我们只需要直接返回顶点颜色即可：

```
fixed4 frag(v2f i) : SV_Target {  
    return fixed4(i.color, 1.0);  
}
```

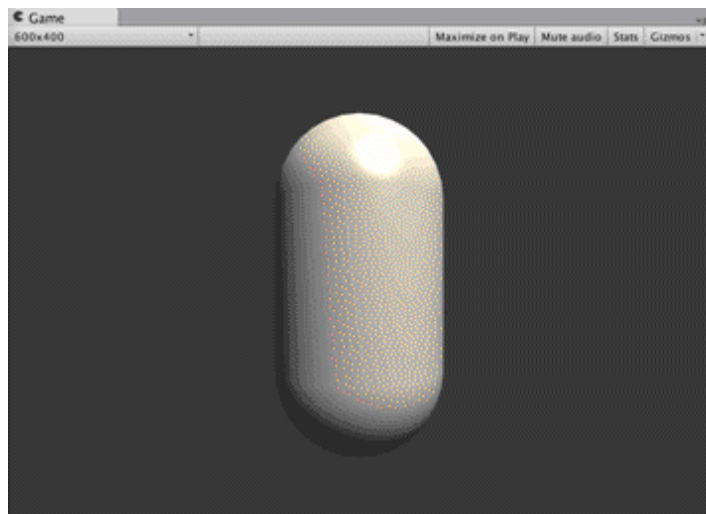
(10) 最后，我们需要把这个Unity Shader的回调Shader设置为内置的Specular：

```
Fallback "Specular"
```

使用逐顶点的方法得到的高光效果有比较大的问题，我们可以在图6.10中看出高光部分明显不平滑。这主要是因为，高光反射部分的计算是非线性的，而在顶点着色器中计算光照再进行插值的过程是线性的，破坏了原计算的非线性关系，就会出现较大的视觉问题。因此，我们就需要使用逐像素的方法来计算高光反射。

### 6.5.2 实践：逐像素光照

我们可以使用逐像素光照来得到更加平滑的高光效果，如图6.11所示。



▲图6.11 逐像素的高光反射光照效果

首先，我们需要进行如下准备工作。

- (1) 使用和6.5.1小节同样的场景。
- (2) 新建一个材质。在本书资源中，该材质名为 `SpecularPixelLevelMat`。
- (3) 新建一个Unity Shader。在本书资源中，该Shader名为 `Chapter6-SpecularPixelLevel`。把新的Shader赋给第2步中创建的材质。
- (4) 把第2步中创建的材质赋给胶囊体。

打开 `Chapter6-SpecularPixelLevel`，删除已有的Shader代码，把上6.5.1节中的代码粘贴进去，并对顶点着色器和片元着色器进行如下修改。

- (1) 修改顶点着色器的输出结构体 `v2f`:

```

struct v2f {
    float4 pos : SV_POSITION;
    float3 worldNormal : TEXCOORD0;
    float3 worldPos : TEXCOORD1;
};

```

(2) 顶点着色器只需要计算世界空间下的法线方向和顶点坐标, 并把它传递给片元着色器即可:

```

v2f vert(a2v v) {
    v2f o;
    // Transform the vertex from object space to projection space
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    // Transform the normal from object space to world space
    o.worldNormal = mul(v.normal, (float3x3)_World2Object);
    // Transform the vertex from object space to world space
    o.worldPos = mul(_Object2World, v.vertex).xyz;

    return o;
}

```

(3) 片元着色器需要计算关键的光照模型:

```

fixed4 frag(v2f i) : SV_Target {
    // Get ambient term
    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;

    fixed3 worldNormal = normalize(i.worldNormal);
    fixed3 worldLightDir = normalize(_WorldSpaceLightPos0.xyz);

    // Compute diffuse term
    fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb *
    saturate(dot(worldNormal, worldLightDir));

    // Get the reflect direction in world space
    fixed3 reflectDir = normalize(reflect(-worldLightDir,
    worldNormal));
    // Get the view direction in world space
    fixed3 viewDir = normalize(_WorldSpaceCameraPos.xyz -
    i.worldPos.xyz);
    // Compute specular term
    fixed3 specular = _LightColor0.rgb * _Specular.rgb *
    pow(saturate(dot(reflectDir, viewDir)), _Gloss);
}

```



```
    return fixed4(ambient + diffuse + specular, 1.0);  
}
```

上面的代码和6.5.1节中的基本相同，在此不再赘述。

可以看出，按逐像素的方式处理光照可以得到更加平滑的高光效果。至此，我们就实现了一个完整的Phong光照模型。

### 6.5.3 Blinn-Phong光照模型

在6.5.2小节中，我们给出了Phong光照模型在Unity中的实现，而在6.2.4节中，我们还提到了另一种高光反射的实现方法——Blinn光照模型。回忆一下，Blinn模型没有使用反射方向，而是引入一个新的矢量 $\hat{\mathbf{h}}$ ，它是通过对视角方向 $\hat{\mathbf{v}}$ 和光照方向 $\hat{\mathbf{l}}$ 相加后再归一化得到的。即

$$\hat{\mathbf{h}} = \frac{\hat{\mathbf{v}} + \hat{\mathbf{l}}}{|\hat{\mathbf{v}} + \hat{\mathbf{l}}|}$$

而Blinn模型计算高光反射的公式如下：

$$\mathbf{c}_{\text{specular}} = (\mathbf{c}_{\text{light}} \cdot \mathbf{m}_{\text{specular}}) \max(0, \hat{\mathbf{n}} \cdot \hat{\mathbf{h}})^{m_{\text{gloss}}}$$

Blinn-Phong模型的实现和6.5.2节中的代码很类似。为此。

(1) 仍然使用和6.5.2节同样的场景。

(2) 新建一个材质。在本书资源中，该材质名为BlinnPhongMat。

(3) 新建一个Unity Shader。在本书资源中，该Shader名为Chapter6-BlinnPhong。把新的Shader赋给第2步中创建的材质。

(4) 把第2步中创建的材质赋给胶囊体。

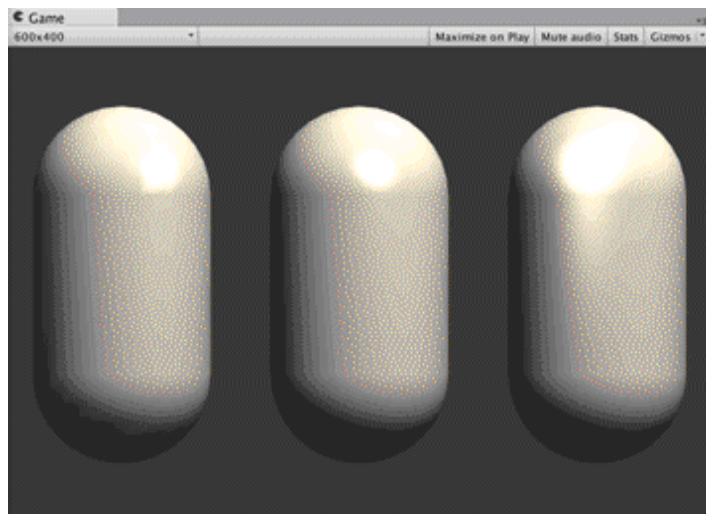
打开Chapter6-BlinnPhong，删除已有的Shader代码，并把6.5.2节中的Chapter6-Specular PixelLevel代码直接粘贴进去。我们只需要修改片元着色器中对高光反射部分的计算代码：

```
fixed4 frag(v2f i) : SV_Target {
    ...

    // Get the view direction in world space
    fixed3 viewDir = normalize(_WorldSpaceCameraPos.xyz -
i.worldPos.xyz);
    // Get the half direction in world space
    fixed3 halfDir = normalize(worldLightDir + viewDir);
    // Compute specular term
    fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0,
dot(worldNormal, halfDir)), _Gloss);

    return fixed4(ambient + diffuse + specular, 1.0);
}
```

图6.12给出了逐顶点的高光反射光照、逐像素的高光反射光照（Phong模型）和Blinn-Phong高光反射光照的对比结果。



▲图6.12 逐顶点的高光反射光照、逐像素的高光反射光照（Phong光照模型）和Blinn-Phong高光反射光照的对比结果

可以看出，**Blinn-Phong**光照模型的高光反射部分看起来更大、更亮一些。在实际渲染中，绝大多数情况我们都会选择**Blinn-Phong**光照模型。需要再次提醒的是，这两种光照模型都是经验模型，也就是说，我们不应该认为**Blinn-Phong**模型是对“正确的”**Phong**模型的近似。实际上，在一些情况下（详见第18章•基于物理的渲染），**Blinn-Phong**模型更符合实验结果。

## 6.6 召唤神龙：使用Unity内置的函数

读者可以发现，在计算光照模型的时候，我们往往需要得到光源方向、视角方向这两个基本信息。在上面的例子中，我们都是自行在代码里计算的，例如使用`normalize(_WorldSpace LightPos0.xyz)`来得到光源方向（这种方法实际只适用于平行光），使用`normalize(_WorldSpace CameraPos.xyz - i.worldPosition.xyz)`来得到视角方向。但如果需要处理更复杂的光照类型，如点光源和聚光灯，我们

计算光源方向的方法就是错误的。这需要我们在代码中先判断光源类型，再计算它的光源信息。具体方法会在9.2节中讲到。

手动计算这些光源信息的过程相对比较麻烦（但并不意味着你不需要了解它们的原理）。幸运的是，Unity提供了一些内置函数来帮助我们计算这些信息。在5.3.1节中，我们给出了UnityCG.cginc里一些非常有用的帮助函数。这里，我们再次回顾一下它们。表6.1给出了计算光照模型时，我们常常使用的一些内置函数。

表6.1      UnityCG.cginc中一些常用的帮助函数

函 数 名	描 述
float3 WorldSpaceViewDir (float4 v)	输入一个模型空间中的顶点位置，返回世界空间中从该点到摄像机的观察方向。内部实现使用了UnityWorldSpaceViewDir函数
float3 UnityWorldSpaceViewDir (float4 v)	输入一个世界空间中的顶点位置，返回世界空间中从该点到摄像机的观察方向
float3 ObjSpaceViewDir (float4 v)	输入一个模型空间中的顶点位置，返回模型空间中从该点到摄像机的观察方向
float3 WorldSpaceLightDir (float4 v)	<b>仅可用于前向渲染中</b> 。输入一个模型空间中的顶点位置，返回世界空间中从该点到光源的光照方向。内部实现使用了UnityWorldSpaceLightDir函数。没有被归一化

函 数 名	描 述
float3 UnityWorldSpaceLightDir (float4 v)	仅可用于前向渲染中。输入一个世界空间中的顶点位置，返回世界空间中从该点到光源的光照方向。没有被归一化
float3 ObjSpaceLightDir (float4 v)	仅可用于前向渲染中。输入一个模型空间中的顶点位置，返回模型空间中从该点到光源的光照方向。没有被归一化
float3 UnityObjectToWorldNormal (float3 norm)	把法线方向从模型空间转换到世界空间中
float3 UnityObjectToWorldDir (float3 dir)	把方向矢量从模型空间变换到世界空间中
float3 UnityWorldToObjectDir(float3 dir)	把方向矢量从世界空间变换到模型空间中

注意，类似UnityXXX的几个函数是Unity 5中新添加的内置函数。这些帮助函数使得我们不需要跟各种变换矩阵、内置变量打交道，也不需要考虑各种不同的情况（例如使用了哪种光源），而仅仅调用一个函数就可以得到需要的信息。上面的9个帮助函数中，有5个我们已经掌握了其内部实现，例如WorldSpaceViewDir函数实现如下：

```
// Computes world space view direction, from object space position
inline float3 UnityWorldSpaceViewDir( in float3 worldPos )
```

```
{  
    return _WorldSpaceCameraPos.xyz - worldPos;  
}
```

可以看出，这与之前计算视角方向的方法一致。**需要注意的是**，这些函数都没有保证得到的方向矢量是单位矢量，因此，我们需要在使用前把它们归一化。

而计算光源方向的3个函数：WorldSpaceLightDir、UnityWorldSpaceLightDir和ObjSpaceLightDir，稍微复杂一些，这是因为，Unity帮我们处理了不同种类光源的情况。**需要注意的是**，这3个函数仅可用于前向渲染（关于什么是前向渲染会在9.1节中讲到）。这是因为只有在前向渲染时，这3个函数里使用的内置变量\_WorldSpaceLightPos0等才会被正确赋值。关于哪些内置变量只会在前向渲染中被正确赋值，可以参见9.1.1节。

下面介绍使用内置函数改写Unity Shader。

我们已经在本节涉及了过多的细节，如果读者无法理解所有内容的话，只需要知道，在实际编写过程中，我们往往会借助于Unity的内置函数来帮助我们进行各种计算，这可以减轻不少我们的“痛苦”。

下面，我们将使用这些内置函数来改写6.5.3小节中使用Blinn-Phong光照模型的Unity Shader。为此。

(1) 在Unity中新建一个场景。在本书资源中，该场景名为Scene\_6\_6。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window -> Lighting -> Skybox中去掉场景中的天空盒子。

(2) 新建一个材质。在本书资源中，该材质名为 **BlinnPhongUseBuildInFunctionMat**。

(3) 新建一个 **Unity Shader**。在本书资源中，该 **Shader** 名为 **Chapter6-BlinnPhongUseBuildIn function**。把新的 **Shader** 赋给第2步中创建的材质。

(4) 创建一个胶囊体，并把第2步中创建的材质赋给它。

**Chapter6-BlinnPhongUseBuildInFunction** 中的代码几乎和 **Chapter6-BlinnPhong** 中的完全一样，只是计算时使用了 **Unity** 的内置函数。修改部分的代码如下：

(1) 在顶点着色器中，我们使用内置的 **UnityObjectToWorldNormal** 函数来计算世界空间下的法线方向：

```
v2f vert(a2v v) {  
    v2f o;  
    ...  
    // Use the build-in function to compute the normal in world  
    space  
    o.worldNormal = UnityObjectToWorldNormal(v.normal);  
    ...  
    return o;  
}
```

(2) 在片元着色器中，我们使用内置的 **UnityWorldSpaceLightDir** 函数和 **UnityWorldSpaceViewDir** 函数来分别计算世界空间的光照方向和视角方向：

```
fixed4 frag(v2f i) : SV_Target {  
    ...  
    fixed3 worldNormal = normalize(i.worldNormal);  
    // Use the build-in function to compute the light direction in
```



```
world space
    // Remember to normalize the result
    fixed3 worldLightDir =
normalize(UnityWorldSpaceLightDir(i.worldPos));

    ...

    // Use the build-in function to compute the view direction in
world space
    // Remember to normalize the result
    fixed3 viewDir = normalize(UnityWorldSpaceViewDir(i.worldPos));
    ...
}
```

需要注意的是，由内置函数得到的方向是没有归一化的，因此我们需要使用**normalize**函数来对结果进行归一化，再进行光照模型的计算。

---

[1] [http://en.wikipedia.org/wiki/Bui\\_Tuong\\_Phong](http://en.wikipedia.org/wiki/Bui_Tuong_Phong)

## 第7章 基础纹理

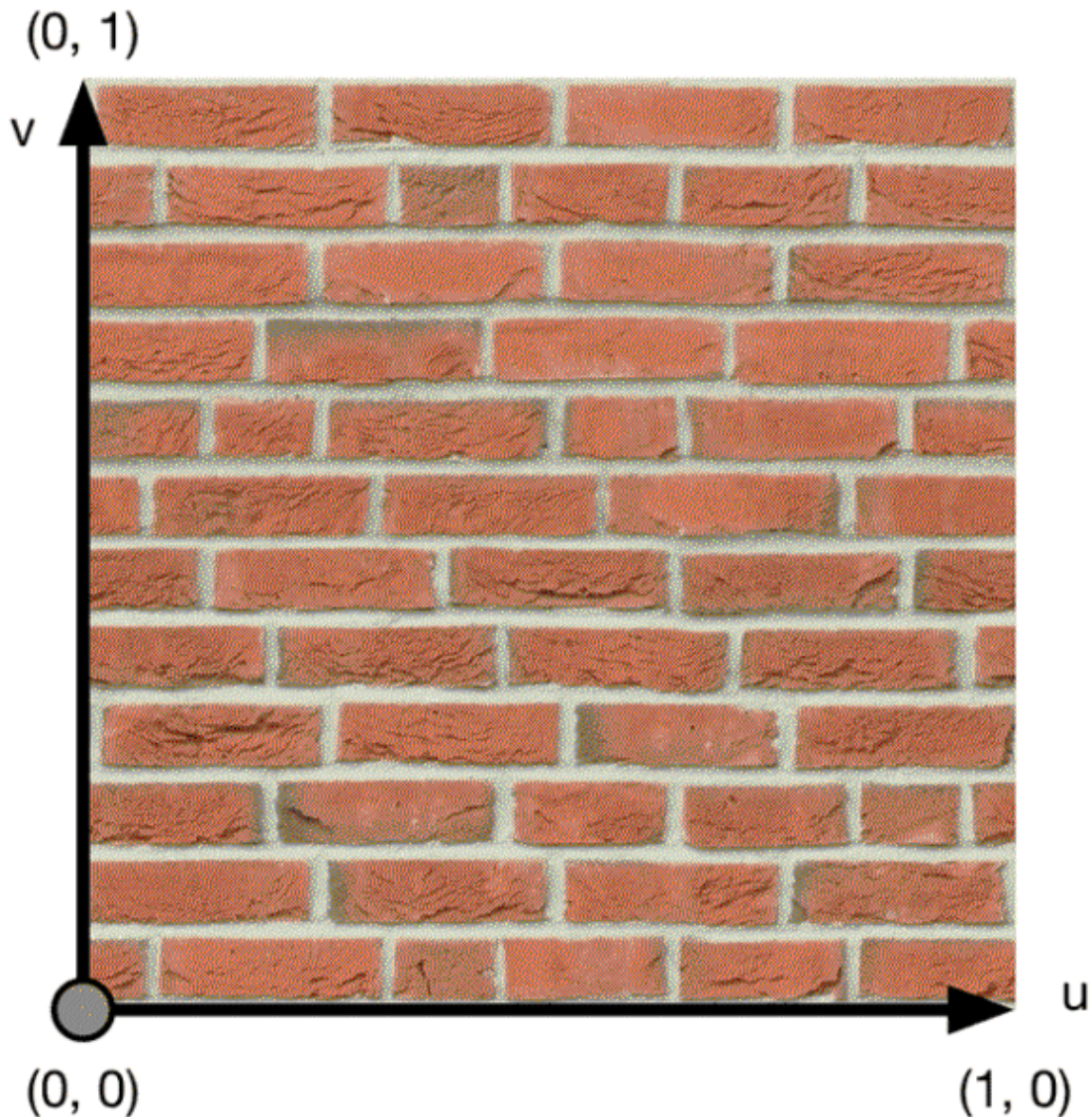
纹理最初的目的就是使用一张图片来控制模型的外观。使用**纹理映射（texture mapping）技术**，我们可以把一张图“黏”在模型表面，**逐纹素（texel）**（纹素的名字是为了和像素进行区分）地控制模型的颜色。

在美术人员建模的时候，通常会在建模软件中利用纹理展开技术把**纹理映射坐标（texture-mapping coordinates）**存储在每个顶点上。纹理映射坐标定义了该顶点在纹理中对应的2D坐标。通常，这些坐标使用一个二维变量( $u, v$ )来表示，其中 $u$ 是横向坐标，而 $v$ 是纵向坐标。因此，纹理映射坐标也被称为UV坐标。

尽管纹理的大小可以是多种多样的，例如可以是 $256 \times 256$ 或者 $1024 \times 1024$ ，但顶点UV坐标的范围通常都被归一化到 $[0, 1]$ 范围内。需要注意的是，纹理采样时使用的纹理坐标不一定是在 $[0, 1]$ 范围内。实际上，这种不在 $[0, 1]$ 范围内的纹理坐标有时会非常有用。与之关系紧密的是纹理的平铺模式，它将决定渲染引擎在遇到不在 $[0, 1]$ 范围内的纹理坐标时如何进行纹理采样。我们将在7.1.2节中更加详细地进行阐述。

在本书之前的章节中，我们曾不止一次地提到过OpenGL和DirectX在二维纹理空间中的坐标系差异问题。重要的事情要说很多次，我们再来回顾一下。在OpenGL里，纹理空间的原点位于左下角，而在DirectX中，原点位于左上角。幸运的是，Unity在绝大多数情况下（特

例情况可以参见5.6节) 为我们处理好了这个差异问题, 也就是说, 即便游戏的目标平台可能既有OpenGL风格的, 也有DirectX风格的, 但我们在Unity中使用的通常只有一种坐标系。Unity使用的纹理空间是符合OpenGL的传统的, 也就是说, 原点位于纹理左下角, 如图7.1所示。



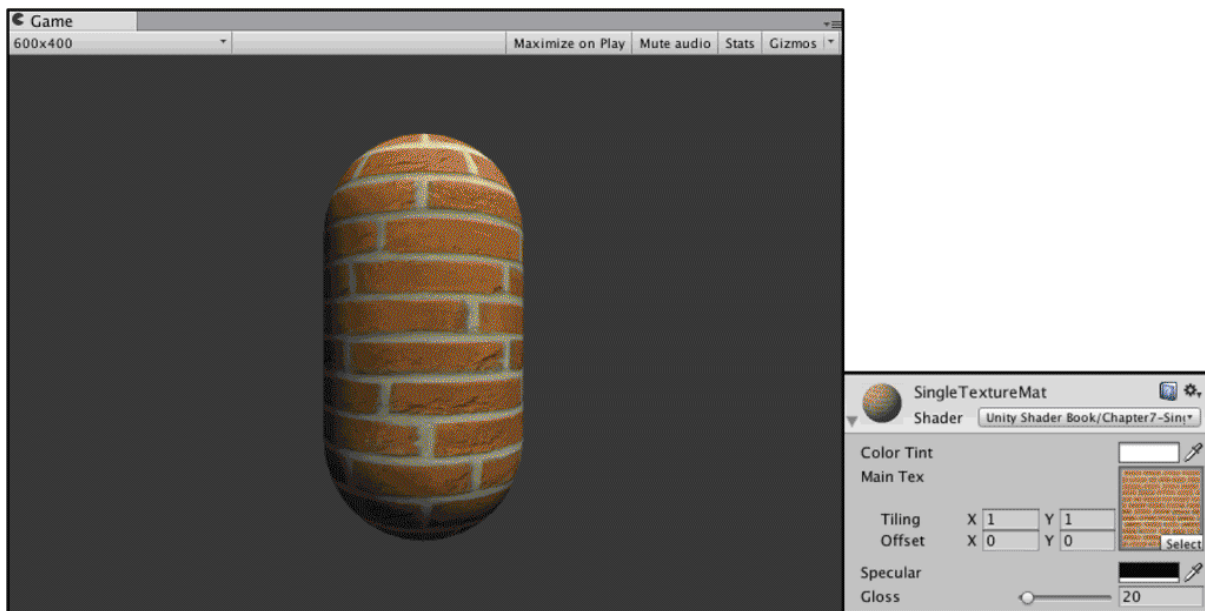
▲ 图7.1 Unity中的纹理坐标

本章将介绍如何在Unity中利用纹理采样来实现更加丰富的视觉效果。在7.1节中，我们将学习如何在Unity Shader中进行最基本的纹理采样，并介绍纹理的属性等基本概念。7.2节将介绍游戏中应用广泛的凹凸纹理，还会解释Unity中法线纹理的一些实现细节。7.3节和7.4节将分别介绍两类特殊的纹理类型，即渐变纹理和遮罩纹理，这些纹理在游戏中的应用非常广泛。

**需要提醒读者注意的是**，本章着重讲述纹理采样的原理，因此实现的Shader往往并不能直接应用到实际项目中（直接使用的话会缺少阴影、光照衰减等效果）。我们会在9.5节给出包含了纹理采样和完整光照模型的可真正使用的Unity Shader。

## 7.1 单张纹理

我们通常会使用一张纹理来代替物体的漫反射颜色。在本节中，我们将学习如何在Unity Shader中使用单张纹理来作为模拟的颜色。在学习完本节后，我们会得到类似图7.2中的效果。



▲ 图7.2 使用单张纹理

### 7.1.1 实践

在本例中，我们仍然使用Blinn-Phong光照模型来计算光照。准备工作如下。

(1) 在Unity中新建一个场景。在本书资源中，该场景名为Scene\_7\_1。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window -> Lighting -> Skybox中去掉场景中的天空盒子。

(2) 新建一个材质。在本书资源中，该材质名为SingleTextureMat。

(3) 新建一个Unity Shader。在本书资源中，该Unity Shader名为Chapter7-SingleTexture。把新的Unity Shader赋给第2步中创建的材质。



(4) 在场景中创建一个胶囊体，并把第2步中的材质赋给该胶囊体。

(5) 保存场景。

打开新建的Chapter7-SingleTexture，删除所有已有代码，并进行如下修改。

(1) 首先，我们需要为这个Unity Shader起一个名字：

```
Shader "Unity Shaders Book/Chapter 7/Single Texture" {
```

(2) 为了使用纹理，我们需要在Properties语义块中添加一个纹理属性：

```
Properties {  
    _Color ("Color Tint", Color) = (1,1,1,1)  
    _MainTex ("Main Tex", 2D) = "white" {}  
    _Specular ("Specular", Color) = (1, 1, 1, 1)  
    _Gloss ("Gloss", Range(8.0, 256)) = 20  
}
```

上面的代码声明了一个名为\_MainTex的纹理，在3.3.2节中，我们已经知道2D是纹理属性的声明方式。我们使用一个字符串后跟一个花括号作为它的初始值，“white”是内置纹理的名字，也就是一个全白的纹理。为了控制物体的整体色调，我们还声明了一个\_Color属性。

(3) 然后，我们在SubShader语义块中定义了一个Pass语义块。而且，我们在Pass的第一行指明了该Pass的光照模式：

```
SubShader {  
    Pass {  
        Tags { "LightMode"="ForwardBase" }  
    }
```

LightMode标签是Pass标签中的一种，它用于定义该Pass在Unity的光照流水线中的角色。

(4) 接着，我们使用CGPROGRAM和ENDCG来包围住Cg代码片，以定义最重要的顶点着色器和片元着色器代码。首先，我们使用#pragma指令来告诉Unity，我们定义的顶点着色器和片元着色器叫什么名字。在本例中，它们的名字分别是vert和frag:

```
CGPROGRAM

#pragma vertex vert
#pragma fragment frag
```

(5) 为了使用Unity内置的一些变量，如\_LightColor0，还需要包含进Unity的内置文件Lighting.cginc:

```
#include "Lighting.cginc"
```

(6) 我们需要在Cg代码片中声明和上述属性类型相匹配的变量，以便和材质面板中的属性建立联系:

```
fixed4 _Color;
sampler2D _MainTex;
float4 _MainTex_ST;
fixed4 _Specular;
float _Gloss;
```

与其他属性类型不同的是，我们还需要为纹理类型的属性声明一个float4类型的变量\_MainTex\_ST。其中，\_MainTex\_ST的名字不是任意起的。在Unity中，我们需要使用**纹理名\_ST**的方式来声明某个纹理的属性。其中，ST是缩放（scale）和平移（translation）的缩写。\_MainTex\_ST可以让我们得到该纹理的缩放和平移（偏移）值，



`_MainTex_ST.xy`存储的是缩放值，而`_MainTex_ST.zw`存储的是偏移值。这些值可以在材质面板的纹理属性中调节，如图7.3所示。在7.1.2节中，我们将更详细地解释这些纹理属性。



▲图7.3 调节纹理的平铺（缩放）和偏移（平移）属性

(7) 接下来，我们需要定义顶点着色器的输入和输出结构体：

```
struct a2v {  
    float4 vertex : POSITION;  
    float3 normal : NORMAL;  
    float4 texcoord : TEXCOORD0;  
};  
  
struct v2f {  
    float4 pos : SV_POSITION;  
    float3 worldNormal : TEXCOORD0;  
    float3 worldPos : TEXCOORD1;  
    float2 uv : TEXCOORD2;  
};
```

在上面的代码中，我们首先在`a2v`结构体中使用`TEXCOORD0`语义声明了一个新的变量`texcoord`，这样Unity就会将模型的第一组纹理坐标存储到该变量中。然后，我们在`v2f`结构体中添加了用于存储纹理坐标的变量`uv`，以便在片元着色器中使用该坐标进行纹理采样。

(8) 然后，我们定义了顶点着色器：

```
v2f vert(a2v v) {  
    v2f o;  
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);  
  
    o.worldNormal = UnityObjectToWorldNormal(v.normal);  
  
    o.worldPos = mul(_Object2World, v.vertex).xyz;
```

```

o.uv = v.texcoord.xy * _MainTex_ST.xy + _MainTex_ST.zw;
// Or just call the built-in function
//      o.uv = TRANSFORM_TEX(v.texcoord, _MainTex);

return o;
}

```

在顶点着色器中，我们使用纹理的属性值 `_MainTex_ST` 来对顶点纹理坐标进行变换，得到最终的纹理坐标。计算过程是，首先使用缩放属性 `_MainTex_ST.xy` 对顶点纹理坐标进行缩放，然后再使用偏移属性 `_MainTex_ST.zw` 对结果进行偏移。Unity 提供了一个内置宏 `TRANSFORM_TEX` 来帮我们计算上述过程。`TRANSFORM_TEX` 是在 `UnityCG.cginc` 中定义的：

```

// Transforms 2D UV by scale/bias property
#define TRANSFORM_TEX(tex,name) (tex.xy * name##_ST.xy +
name##_ST.zw)

```

它接受两个参数，第一个参数是顶点纹理坐标，第二个参数是纹理名，在它的实现中，将利用 **纹理名\_ST** 的方式来计算变换后的纹理坐标。

(9) 我们还需要实现片元着色器，并在计算漫反射时使用纹理中的纹素质：

```

fixed4 frag(v2f i) : SV_Target {
    fixed3 worldNormal = normalize(i.worldNormal);
    fixed3 worldLightDir =
normalize(UnityWorldSpaceLightDir(i.worldPos));

    // Use the texture to sample the diffuse color
    fixed3 albedo = tex2D(_MainTex, i.uv).rgb * _Color.rgb;

    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;

    fixed3 diffuse = _LightColor0.rgb * albedo * max(0,
dot(worldNormal, worldLightDir));
}

```

```
fixed3 viewDir = normalize(UnityWorldSpaceViewDir(i.worldPos));
fixed3 halfDir = normalize(worldLightDir + viewDir);
fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0,
dot(worldNormal, halfDir)), _Gloss);

return fixed4(ambient + diffuse + specular, 1.0);
}
```

上面的代码首先计算了世界空间下的法线方向和光照方向。然后，使用Cg的tex2D函数对纹理进行采样。它的第一个参数是需要被采样的纹理，第二个参数是一个float2类型的纹理坐标，它将返回计算得到的纹素值。我们使用采样结果和颜色属性\_Color的乘积来作为材质的反射率albedo，并把它和环境光照相乘得到环境光部分。随后，我们使用albedo来计算漫反射光照的结果，并和环境光照、高光反射光照相加后返回。

(10) 最后，我们为该Shader设置了合适的Fallback:

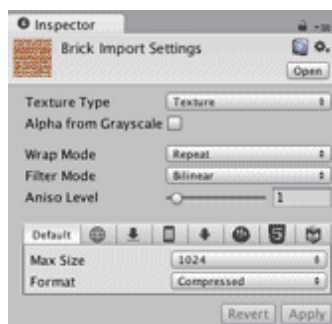
```
Fallback "Specular"
```

保存后返回Unity中查看。在SingleTextureMat的面板上，我们使用本书资源中的Brick\_Diffuse.jpg纹理对Main Tex属性进行赋值。

### 7.1.2 纹理的属性

虽然很多资料把Unity的纹理映射描述得很简单——声明一个纹理变量，再使用tex2D函数采样。实际上，在渲染流水线中，纹理映射的实现远比我们想象的复杂。在本书不会过多涉及一些具体的实现细节，但要解释一些我们认为读者必须要知道的事情。在本节中，我们将关注Unity中的纹理属性。

在我们向Unity中导入一张纹理资源后，可以在它的材质面板上调整其属性，如图7.4所示。



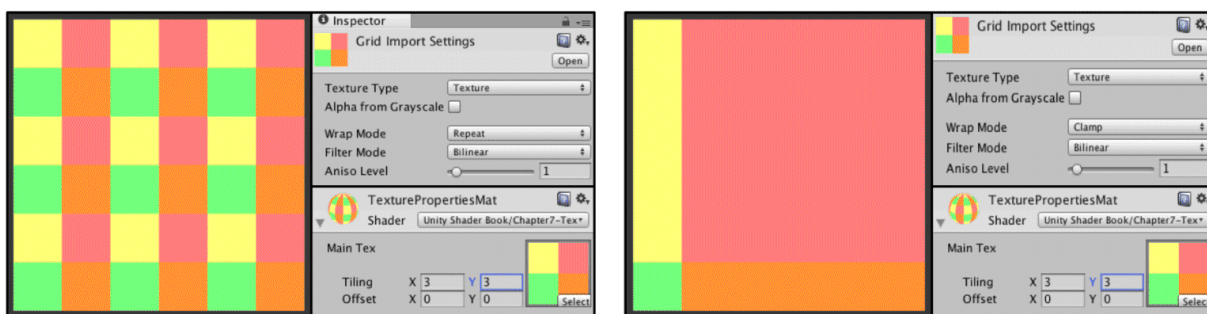
▲ 图7.4 纹理的属性

纹理面板中的第一个属性是纹理类型。在本节中，我们使用的是 *Texture* 类型，在下面的法线纹理一节中，我们会使用 *Normal map* 类型。而在后面的章节中，我们还会看到 *Cubemap* 等高级纹理类型。我们之所以要为导入的纹理选择合适的类型，是因为只有这样才能让Unity知道我们的意图，为Unity Shader传递正确的纹理，并在一些情况下可以让Unity对该纹理进行优化。

当把纹理类型设置成 *Texture* 后，下面会有一个 *Alpha from Grayscale* 复选框，如果勾选了它，那么透明通道的值将会由每个像素的灰度值生成。关于透明效果，我们会在第8章中讲到。在这里我们不需要勾选它。

下面一个属性非常重要——*Wrap Mode*。它决定了当纹理坐标超过  $[0, 1]$  范围后将会如何被平铺。*Wrap Mode* 有两种模式：一种是 *Repeat*，在这种模式下，如果纹理坐标超过了1，那么它的整数部分将会被舍弃，而直接使用小数部分进行采样，这样的结果是纹理将会不断重

复；另一种是`Clamp`，在这种模式下，如果纹理坐标大于1，那么将会截取到1，如果小于0，那么将会截取到0。图7.5给出了两种模式下平铺一张纹理的效果（读者可在本书资源中的Scene\_7\_1\_2\_a中找到相应场景）。



▲ 图7.5 Wrap Mode决定了当纹理坐标超过[0, 1]范围后将会如何被平铺

图7.5展示了在纹理的平铺（`Tiling`）属性为(3, 3)时分别使用两种Wrap Mode的结果。左图使用了Repeat模式，在这种模式下纹理将会不断重复；右图使用了Clamp模式，在这种模式下超过范围的部分将会截取到边界值，形成一个条形结构。

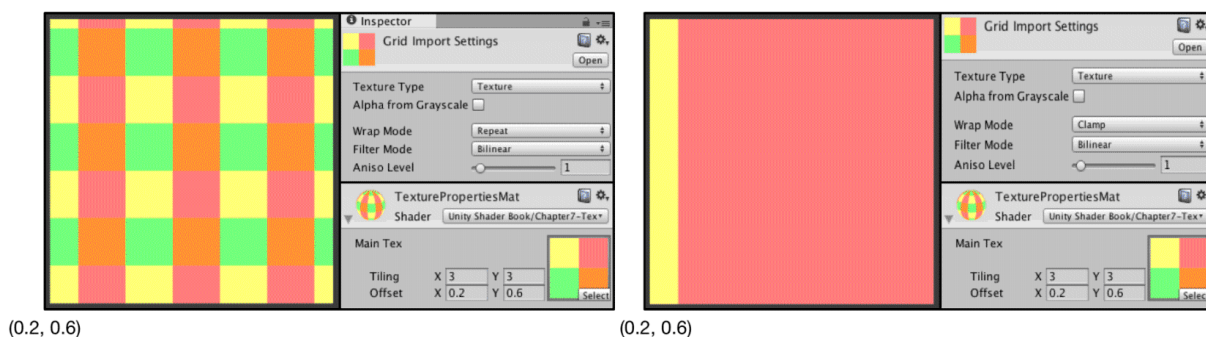
需要注意的是，想要让纹理得到这样的效果，我们必须使用纹理的属性（例如上面的`_MainTex_ST`变量）在Unity Shader中对顶点纹理坐标进行相应的变换。也就是说，代码中需要包含类似下面的代码：

```
o.uv = v.texcoord.xy * _MainTex_ST.xy + _MainTex_ST.zw;  
// Or just call the built-in function  
o.uv = TRANSFORM_TEX(v.texcoord, _MainTex);
```

我们还可以在材质面板中调整纹理的偏移量，图7.6给出了两种模式下调整纹理偏移量的一个例子。

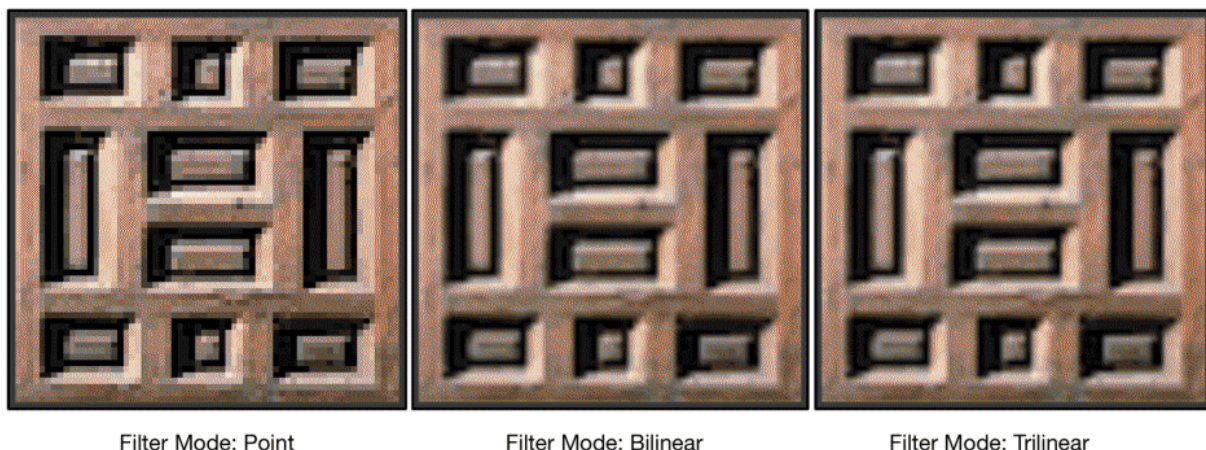
图7.6展示了在纹理的偏移属性为(0.2, 0.6)时分别使用两种Wrap Mode的结果，左图使用了Repeat模式，右图使用了Clamp模式。

纹理导入面板中的下一个属性是**Filter Mode**属性，它决定了当纹理由于变换而产生拉伸时将会采用哪种滤波模式。**Filter Mode**支持3种模式：*Point*，*Bilinear*以及*Trilinear*。它们得到的图片滤波效果依次提升，但需要耗费的性能也依次增大。纹理滤波会影响放大或缩小纹理时得到的图片质量。例如，当我们把一张64×64大小的纹理贴在一个512×512大小的平面上时，就需要放大纹理。图7.7给出了3种滤波模式下的放大结果。读者可以在本书资源中的Scene\_7\_1\_2\_b中找到该场景。



▲ 图7.6 偏移（Offset）属性决定了纹理坐标的偏移量





▲ 图7.7 在放大纹理时，分别使用3种Filter Mode得到的结果

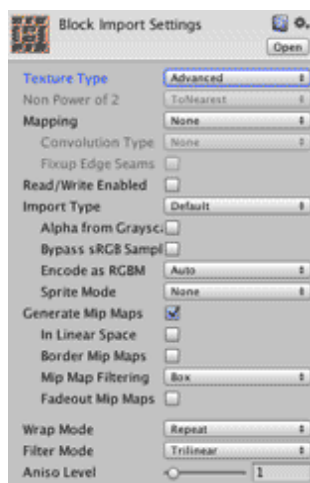
纹理缩小的过程比放大更加复杂一些，此时原纹理中的多个像素将会对应一个目标像素。纹理缩小更加复杂的原因在于我们往往需要处理抗锯齿问题，一个最常使用的方法就是使用**多级渐远纹理**

**(mipmapping)** 技术。其中“mip”是拉丁文“multum in parvo”的缩写，它的意思是“在一个小空间中有许多东西”。如同它的名字，多级渐远纹理技术将原纹理提前用滤波处理来得到很多更小的图像，形成了一个图像金字塔，每一层都是对上一层图像降采样的结果。这样在实时运行时，就可以快速得到结果像素，例如当物体远离摄像机时，可以直接使用较小的纹理。但缺点是需要使用一定的空间用于存储这些多级渐远纹理，通常会多占用33%的内存空间。这是一种典型的用空间换取时间的方法。在Unity中，我们可以在纹理导入面板中，首先将纹理类型（Texture Type）选择成*Advanced*，再勾选*Generate Mip Maps*即可开启多级渐远纹理技术。同时，我们还可以选择生成多级渐远纹理时是否使用线性空间（用于伽玛校正，详见18.4.2节）以及采用的滤波器等，如图7.8所示。

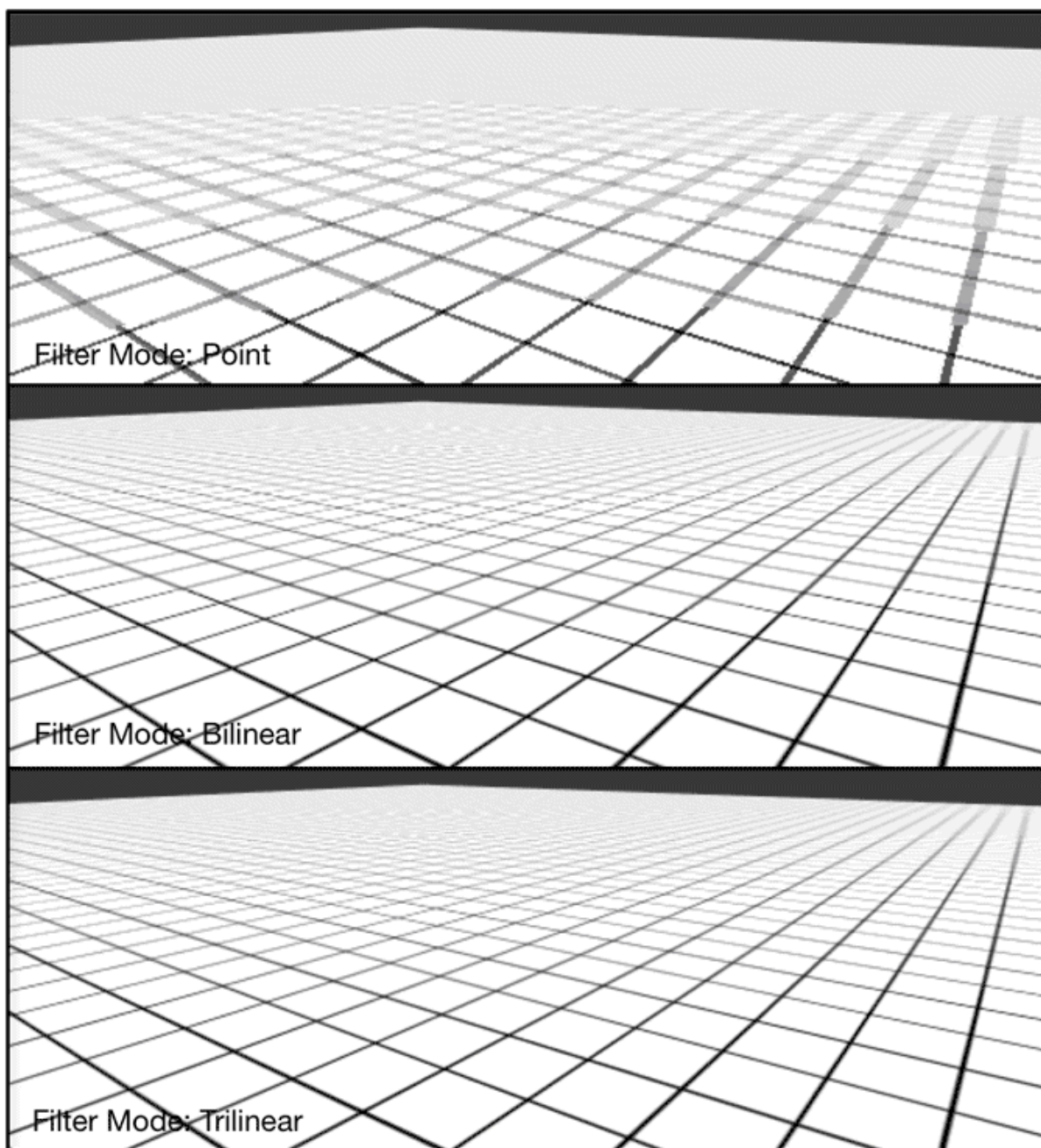


图7.9给出了从一个倾斜的角度观察一个网格结构的地板时，使用不同Filter Mode（同时也使用了多级渐远纹理技术）得到的效果。读者可以在本书资源中的Scene\_7\_1\_2\_c中找到该场景。

在内部实现上，Point模式使用了**最近邻（nearest neighbor）**滤波，在放大或缩小时，它的采样像素数目通常只有一个，因此图像会看起来有种像素风格的效果。而Bilinear滤波则使用了线性滤波，对于每个目标像素，它会找到4个邻近像素，然后对它们进行线性插值混合后得到最终像素，因此图像看起来像被模糊了。而Trilinear滤波几乎是和Bilinear一样的，只是Trilinear还会在多级渐远纹理之间进行混合。如果一张纹理没有使用多级渐远纹理技术，那么Trilinear得到的结果是和Bilinear就一样的。通常，我们会选择Bilinear滤波模式。需要注意的是，有时我们不希望纹理看起来是模糊的，例如对于一些类似棋盘的纹理，我们希望它就是像素风的，这时我们可能会选择Point模式。



▲ 图7.8 在Advanced模式下可以设置多级渐远纹理的相关属性



▲ 图7.9 从上到下： Point滤波 + 多级渐远纹理技术， Bilinear滤波 + 多级渐远纹理技术）， Trilinear滤波 + 多级渐远纹理技术

最后，我们来讲一下纹理的最大尺寸和纹理模式。当我们在为不同平台发布游戏时，需要考虑目标平台的纹理尺寸和质量问题。Unity

允许我们为不同目标平台选择不同的分辨率，如图7.10所示。



▲ 图7.10 选择纹理的最大尺寸和纹理模式

如果导入的纹理大小超过了*Max Texture Size*中的设置值，那么Unity将会把该纹理缩放为这个最大分辨率。理想情况下，导入的纹理可以是非正方形的，但长宽的大小应该是2的幂，例如2、4、8、16、32、64等。如果使用了非2的幂大小（Non Power of Two, NPOT）的纹理，那么这些纹理往往会占用更多的内存空间，而且GPU读取该纹理的速度也会有所下降。有一些平台甚至不支持这种NPOT纹理，这时Unity在内部会把它缩放成最近的2的幂大小。出于性能和空间的考虑，我们应该尽量使用2的幂大小的纹理。

而*Format*决定了Unity内部使用哪种格式来存储该纹理。如果我们将*Texture Type*设置为*Advanced*，那么会有更多的*Format*供我们选择。这里不再依次介绍每种纹理模式，但需要知道的是，使用的纹理格式精度越高（例如使用*Truecolor*），占用的内存空间越大，但得到的效果也越好。我们可以从纹理导入面板的最下方看到存储该纹理需要占

用的内存空间（如果开启了多级渐远纹理技术，也会增加纹理的内存占用）。当游戏使用了大量Truecolor类型的纹理时，内存可能会迅速增加，因此对于一些不需要使用很高精度的纹理（例如用于漫反射颜色的纹理），我们应该尽量使用压缩格式。

## 7.2 凹凸映射

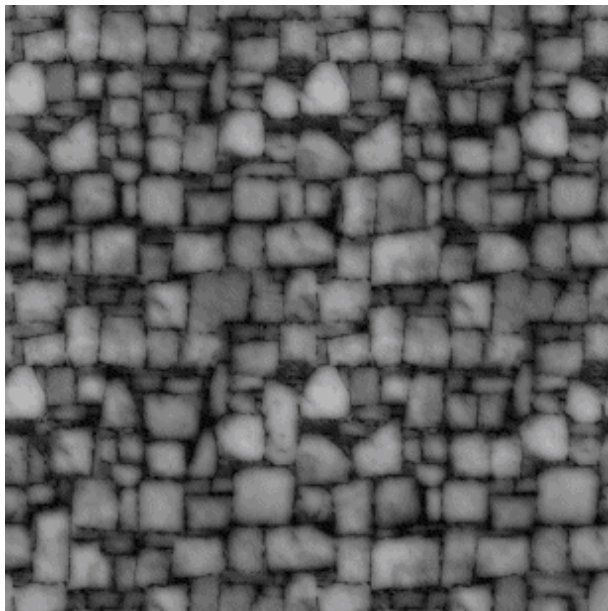
纹理的另一种常见的应用就是**凹凸映射（bump mapping）**。凹凸映射的目的是使用一张纹理来修改模型表面的法线，以便为模型提供更多的细节。这种方法不会真的改变模型的顶点位置，只是让模型看起来好像是“凹凸不平”的，但可以从模型的轮廓处看出“破绽”。

有两种主要的方法可以用来进行凹凸映射：一种方法是使用一张**高度纹理（height map）**来模拟**表面位移（displacement）**，然后得到一个修改后的法线值，这种方法也被称为**高度映射（height mapping）**；另一种方法则是使用一张**法线纹理（normal map）**来直接存储表面法线，这种方法又被称为**法线映射（normal mapping）**。尽管我们常常将凹凸映射和法线映射当成是相同的技术，但读者需要知道它们之间的不同。

### 7.2.1 高度纹理

我们首先来看第一种技术，即使用一张高度图来实现凹凸映射。高度图中存储的是强度值（**intensity**），它用于表示模型表面局部的海拔高度。因此，颜色越浅表明该位置的表面越向外凸起，而颜色越深表明该位置越向里凹。这种方法的好处是非常直观，我们可以从高度图中明确地知道一个模型表面的凹凸情况，但缺点是计算更加复杂，

在实时计算时不能直接得到表面法线，而是需要由像素的灰度值计算而得，因此需要消耗更多的性能。图7.11给出了一张高度图。



▲图7.11 高度图

高度图通常会和法线映射一起使用，用于给出表面凹凸的额外信息。也就是说，我们通常会使用法线映射来修改光照。

### 7.2.2 法线纹理

而法线纹理中存储的就是表面的法线方向。由于法线方向的分量范围在 $[-1, 1]$ ，而像素的分量范围为 $[0, 1]$ ，因此我们需要做一个映射，通常使用的映射就是：

这就要求，我们在Shader中对法线纹理进行纹理采样后，还需要对结果进行一次反映射的过程，以得到原先的法线方向。反映射的过程

实际就是使用上面映射函数的逆函数：

$$normal = pixel \times 2 - 1$$

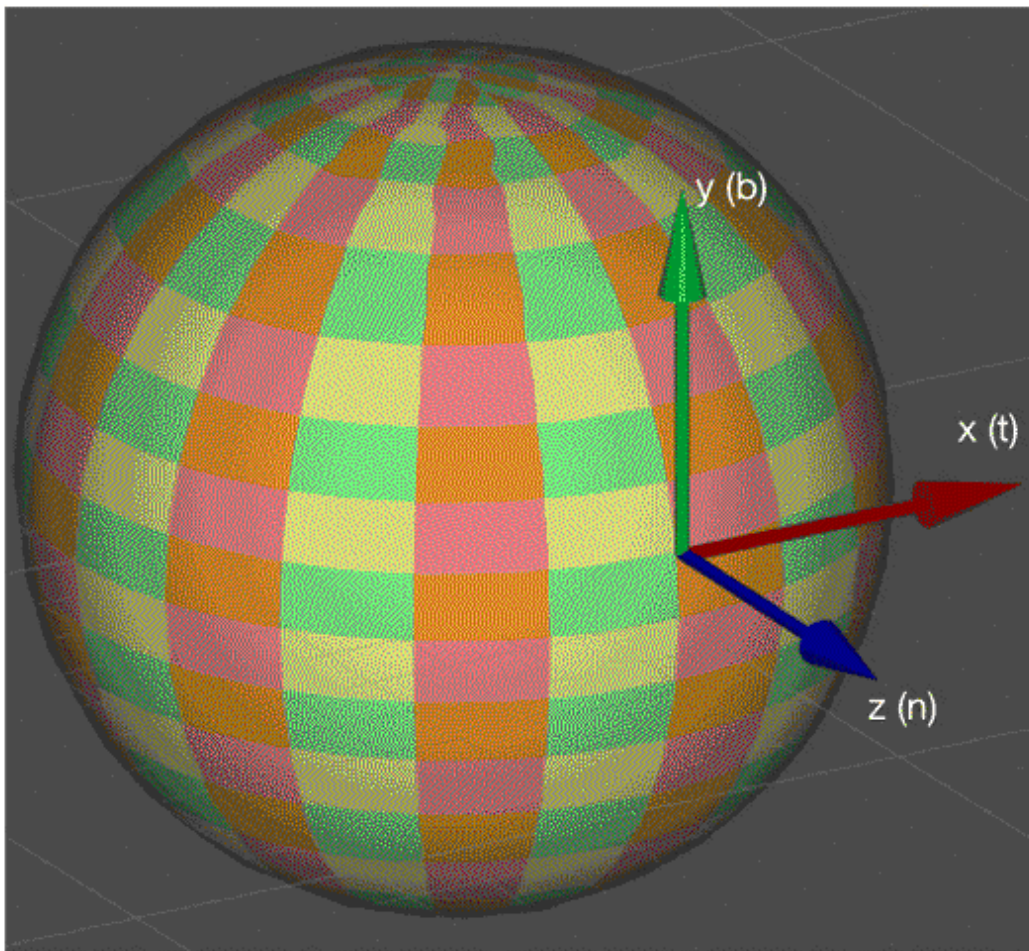
然而，由于方向是相对于坐标空间来说的，那么法线纹理中存储的法线方向在哪个坐标空间中呢？对于模型顶点自带的法线，它们是定义在模型空间中的，因此一种直接的想法就是将修改后的模型空间中的表面法线存储在一张纹理中，这种纹理被称为是**模型空间的法线纹理（object-space normal map）**。然而，在实际制作中，我们往往会采用另一种坐标空间，即模型顶点的**切线空间（tangent space）**来存储法线。对于模型的每个顶点，它都有一个属于自己的切线空间，这个切线空间的原点就是该顶点本身，而z轴是顶点的法线方向（ $n$ ），x轴是顶点的切线方向（ $t$ ），而y轴可由法线和切线叉积而得，也被称为是副切线（bitangent,  $b$ ）或副法线，如图7.12所示。

这种纹理被称为是**切线空间的法线纹理（tangent-space normal map）**。图7.13分别给出了模型空间和切线空间下的法线纹理（图片来源：<http://www.surlybird.com/tutorials/TangentSpace/>）。

从图7.13中可以看出，模型空间下的法线纹理看起来是“五颜六色”的。这是因为所有法线所在的坐标空间是同一个坐标空间，即模型空间，而每个点存储的法线方向是各异的，有的是 $(0, 1, 0)$ ，经过映射后存储到纹理中就对应了 $RGB(0.5, 1, 0.5)$ 浅绿色，有的是 $(0, -1, 0)$ ，经过映射后存储到纹理中就对应了 $(0.5, 0, 0.5)$ 紫色。而切线空间下的法线纹理看起来几乎全部是浅蓝色的。这是因为，每个法线方向所在的坐标空间是不一样的，即是表面每点各自的切线空间。这种法线纹理其实就是存储了每个点在各自的切线空间中的法线扰动方向。也就是

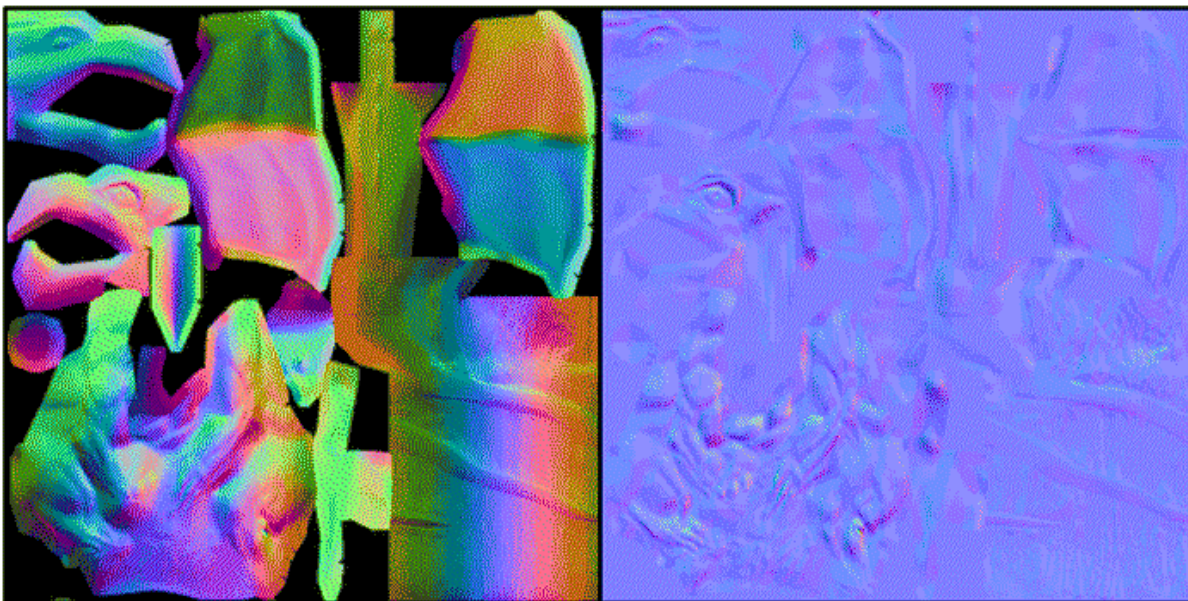


说，如果一个点的法线方向不变，那么在它的切线空间中，新的法线方向就是 $z$ 轴方向，即值为 $(0, 0, 1)$ ，经过映射后存储在纹理中就对应了 $RGB(0.5, 0.5, 1)$ 浅蓝色。而这个颜色就是法线纹理中大片的蓝色。这些蓝色实际上说明顶点的大部分法线是和模型本身法线一样的，不需要改变。



▲图7.12 模型顶点的切线空间。其中，原点对应了顶点坐标， $x$ 轴是切线方向 ( $t$ )， $y$ 轴是副切线方向 ( $b$ )， $z$ 轴是法线方向 ( $n$ )





▲图7.13 左边：模型空间下的法线纹理。右边：切线空间下的法线纹理

总体来说，模型空间下的法线纹理更符合人类的直观认识，而且法线纹理本身也很直观，容易调整，因为不同的法线方向就代表了不同的颜色。但美术人员往往更喜欢使用切线空间下的法线纹理。那么，为什么他们更偏好使用这个看起来“很蹩脚”的切线空间呢？

实际上，法线本身存储在哪个坐标系中都是可以的，我们甚至可以选择存储在世界空间下。但问题是，我们并不是单纯地想要得到法线，后续的光照计算才是我们的目的。而选择哪个坐标系意味着我们需要把不同信息转换到相应的坐标系中。例如，如果选择了切线空间，我们需要把从法线纹理中得到的法线方向从切线空间转换到世界空间（或其他空间）中。

总体来说，使用模型空间来存储法线的优点如下。

- 实现简单，更加直观。我们甚至都不需要模型原始的法线和切线等信息，也就是说，计算更少。生成它也非常简单，而如果要生成切线空间下的法线纹理，由于模型的切线一般是和UV方向相同，因此想要得到效果比较好的法线映射就要求纹理映射也是连续的。
- 在纹理坐标的缝合处和尖锐的边角部分，可见的突变（缝隙）较少，即可以提供平滑的边界。这是因为模型空间下的法线纹理存储的是同一坐标系下的法线信息，因此在边界处通过插值得到的法线可以平滑变换。而切线空间下的法线纹理中的法线信息是依靠纹理坐标的方向得到的结果，可能会在边缘处或尖锐的部分造成更多可见的缝合迹象。

但使用切线空间有更多优点。

- 自由度很高。模型空间下的法线纹理记录的是**绝对法线信息**，仅可用于创建它时的那个模型，而应用到其他模型上效果就完全错误了。而切线空间下的法线纹理记录的是相对法线信息，这意味着，即便把该纹理应用到一个完全不同的网格上，也可以得到一个合理的结果。
- 可进行UV动画。比如，我们可以移动一个纹理的UV坐标来实现一个凹凸移动的效果，但使用模型空间下的法线纹理会得到完全错误的结果。原因同上。这种UV动画在水或者火山熔岩这种类型的物体上会经常用到。
- 可以重用法线纹理。比如，一个砖块，我们仅使用一张法线纹理就可以用到所有的6个面上。原因同上。

- 可压缩。由于切线空间下的法线纹理中法线的 $Z$ 方向总是正方向，因此我们可以仅存储 $XY$ 方向，而推导得到 $Z$ 方向。而模型空间下的法线纹理由于每个方向都是可能的，因此必须存储3个方向的值，不可压缩。

切线空间下的法线纹理的前两个优点足以让很多人放弃模型空间下的法线纹理而选择它。从上面的优点可以看出，切线空间在很多情况下都优于模型空间，而且可以节省美术人员的工作。因此，在本书中，我们使用的也是切线空间下的法线纹理。

### 7.2.3 实践

我们需要在计算光照模型中统一各个方向矢量所在的坐标空间。由于法线纹理中存储的法线是切线空间下的方向，因此我们通常有两种选择：一种选择是在切线空间下进行光照计算，此时我们需要把光照方向、视角方向变换到切线空间下；另一种选择是在世界空间下进行光照计算，此时我们需要把采样得到的法线方向变换到世界空间下，再和世界空间下的光照方向和视角方向进行计算。从效率上来说，第一种方法往往要优于第二种方法，因为我们可以顶点着色器中就完成对光照方向和视角方向的变换，而第二种方法由于要先对法线纹理进行采样，所以变换过程必须在片元着色器中实现，这意味着我们需要在片元着色器中进行一次矩阵操作。但从通用性角度来说，第二种方法要优于第一种方法，因为有时我们需要在世界空间下进行一些计算，例如在使用Cubemap进行环境映射时，我们需要使用世界空间下的反射方向对Cubemap进行采样。如果同时需要进行法线映射，我们就需要把法线方向变换到世界空间下。当然，读者可以选择其他坐

标空间进行计算，例如模型空间等，但切线空间和世界空间是最为常用的两种空间。在本节中，我们将依次实现上述的两种方法。

## 1. 在切线空间下计算

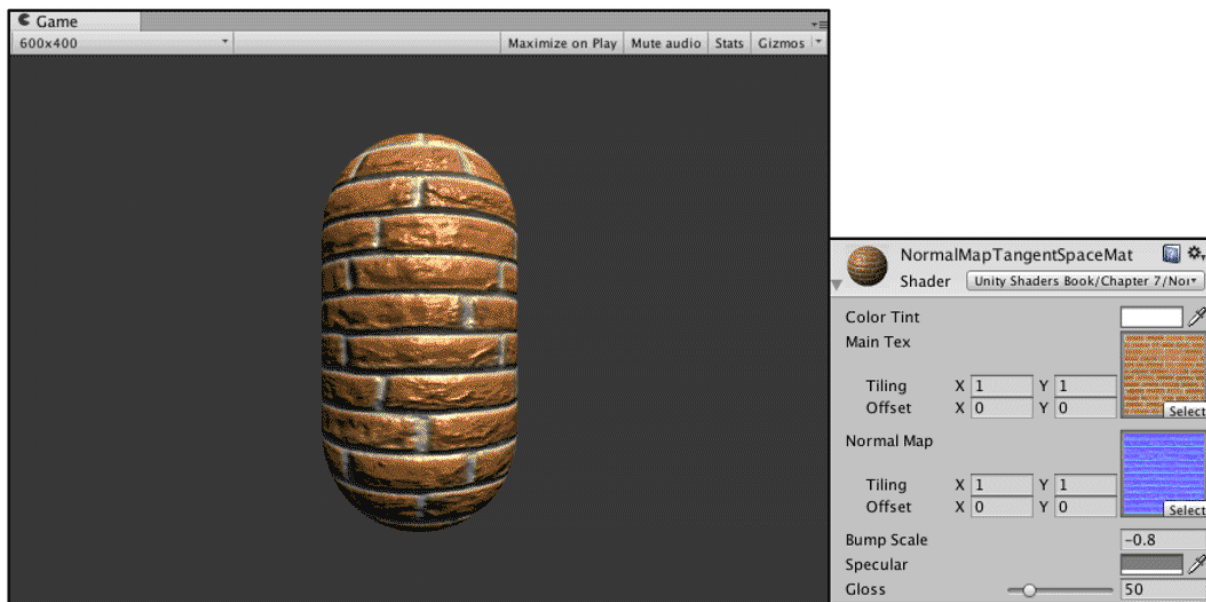
我们首先来实现第一种方法，即在切线空间下计算光照模型。基本思路是：在片元着色器中通过纹理采样得到切线空间下的法线，然后再与切线空间下的视角方向、光照方向等进行计算，得到最终的光照结果。为此，我们首先需要在顶点着色器中把视角方向和光照方向从模型空间变换到切线空间中，即我们需要知道从模型空间到切线空间的变换矩阵。这个变换矩阵的逆矩阵，即从切线空间到模型空间的变换矩阵是非常容易求得的，我们在顶点着色器中按切线（x轴）、副切线（y轴）、法线（z轴）的顺序**按列**排列即可得到（数学原理详见4.6.2节）。在4.6.2节中我们已经知道，如果一个变换中仅存在平移和旋转变换，那么这个变换的逆矩阵就等于它的转置矩阵，而从切线空间到模型空间的变换正是符合这样要求的变换。因此，从模型空间到切线空间的变换矩阵就是从切线空间到模型空间的变换矩阵的转置矩阵，我们把切线（x轴）、副切线（y轴）、法线（z轴）的顺序**按行**排列即可得到。在本节最后，我们可以得到类似图7.14中的效果。

为此，我们进行如下准备工作。

（1）在Unity中新建一个场景。在本书资源中，该场景名为Scene\_7\_2\_3。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window -> Lighting -> Skybox中去掉场景中的天空盒子。

(2) 新建一个材质。在本书资源中，该材质名为 NormalMapTangentSpaceMat。

(3) 新建一个Unity Shader。在本书资源中，该Unity Shader名为 Chapter7-NormalMapTangentSpace。把新的Unity Shader赋给第2步中创建的材质。



▲图7.14 使用法线纹理

(4) 在场景中创建一个胶囊体，并把第2步中的材质赋给该胶囊体。

(5) 保存场景。

打开新建的Chapter7-NormalMapTangentSpace，删除所有已有代码，并进行如下修改。

(1) 首先，我们为该Unity Shader定义一个名字：

```
Shader "Unity Shaders Book/Chapter 7/Normal Map In Tangent Space" {
```

(2) 然后，我们在**Properties**语义块中添加了法线纹理的属性，以及用于控制凹凸程度的属性：

```
Properties {  
    _Color ("Color Tint", Color) = (1,1,1,1)  
    _MainTex ("Main Tex", 2D) = "white" {}  
    _BumpMap ("Normal Map", 2D) = "bump" {}  
    _BumpScale ("Bump Scale", Float) = 1.0  
    _Specular ("Specular", Color) = (1, 1, 1, 1)  
    _Gloss ("Gloss", Range(8.0, 256)) = 20  
}
```

对于法线纹理**\_BumpMap**，我们使用**"bump"**作为它的默认值。**"bump"**是Unity内置的法线纹理，当没有提供任何法线纹理时，**"bump"**就对应了模型自带的法线信息。**\_BumpScale**则是用于控制凹凸程度的，当它为**0**时，意味着该法线纹理不会对光照产生任何影响。

(3) 我们在**SubShader**语义块中定义了一个**Pass**语义块，并且在**Pass**的第一行指明了该**Pass**的光照模式：

```
SubShader {  
    Pass {  
        Tags { "LightMode"="ForwardBase" }  
    }  
}
```

**LightMode**标签是**Pass**标签中的一种，它用于定义该**Pass**在Unity的光照流水线中的角色。

(4) 接着，我们使用**CGPROGRAM**和**ENDCG**来包围住**Cg**代码片，以定义最重要的顶点着色器和片元着色器代码。首先，我们使用



`#pragma`指令来告诉Unity，我们定义的顶点着色器和片元着色器叫什么名字。在本例中，它们的名字分别是`vert`和`frag`：

```
CGPROGRAM

#pragma vertex vert
#pragma fragment frag
```

(5) 为了使用Unity内置的一些变量，如`_LightColor0`，还需要包含进Unity的内置文件`Lighting.cginc`：

```
#include "Lighting.cginc"
```

(6) 为了和`Properties`语义块中的属性建立联系，我们在Cg代码块中声明了和上述属性类型匹配的变量：

```
fixed4 _Color;
sampler2D _MainTex;
float4 _MainTex_ST;
sampler2D _BumpMap;
float4 _BumpMap_ST;
float _BumpScale;
fixed4 _Specular;
float _Gloss;
```

为了得到该纹理的属性（平铺和偏移系数），我们为`_MainTex`和`_BumpMap`定义了`_MainTex_ST`和`_BumpMap_ST`变量。

(7) 我们已经知道，切线空间是由顶点法线和切线构建出的一个坐标空间，因此我们需要得到顶点的切线信息。为此，我们修改顶点着色器的输入结构体`a2v`：

```
struct a2v {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
    float4 tangent : TANGENT;
```



```
float4 texcoord : TEXCOORD0;  
};
```

我们使用TANGENT语义来描述float4类型的tangent变量，以告诉Unity把顶点的切线方向填充到tangent变量中。需要注意的是，和法线方向normal不同，tangent的类型是float4，而非float3，这是因为我们需要使用tangent.w分量来决定切线空间中的第三个坐标轴——副切线的方向性。

(8) 我们需要在顶点着色器中计算切线空间下的光照和视角方向，因此我们在v2f结构体中添加了两个变量来存储变换后的光照和视角方向：

```
struct v2f {  
    float4 pos : SV_POSITION;  
    float4 uv : TEXCOORD0;  
    float3 lightDir: TEXCOORD1;  
    float3 viewDir : TEXCOORD2;  
};
```

(9) 定义顶点着色器：

```
v2f vert(a2v v) {  
    v2f o;  
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);  
  
    o.uv.xy = v.texcoord.xy * _MainTex_ST.xy + _MainTex_ST.zw;  
    o.uv.zw = v.texcoord.xy * _BumpMap_ST.xy + _BumpMap_ST.zw;  
  
    // Compute the binormal  
    // float3 binormal = cross( normalize(v.normal),  
    normalize(v.tangent.xyz) ) * v.tangent.w;  
    // // Construct a matrix which transform vectors from object space  
    to tangent space  
    // float3x3 rotation = float3x3(v.tangent.xyz, binormal,  
    v.normal);  
    // Or just use the built-in macro  
    TANGENT_SPACE_ROTATION;  
  
    // Transform the light direction from object space to tangent
```

```

space
    o.lightDir = mul(rotation, ObjSpaceLightDir(v.vertex)).xyz;
    // Transform the view direction from object space to tangent
space
    o.viewDir = mul(rotation, ObjSpaceViewDir(v.vertex)).xyz;

    return o;
}

```

由于我们使用了两张纹理，因此需要存储两个纹理坐标。为此，我们把v2f中的uv变量的类型定义为float4类型，其中xy分量存储了\_MainTex的纹理坐标，而zw分量存储了\_BumpMap的纹理坐标（实际上，\_MainTex和\_BumpMap通常会使用同一组纹理坐标，出于减少插值寄存器的使用数目的目的，我们往往只计算和存储一个纹理坐标即可）。然后，我们把模型空间下切线方向、副切线方向和法线方向按行排列来得到从模型空间到切线空间的变换矩阵rotation。需要注意的是，在计算副切线时我们使用v.tangent.w和叉积结果进行相乘，这是因为和切线与法线方向都垂直的方向有两个，而w决定了我们选择其中一个方向。Unity也提供了一个内置宏TANGENT\_SPACE\_ROTATION（在UnityCG.cginc中被定义）来帮助我们直接计算得到rotation变换矩阵，它的实现和上述代码完全一样。然后，我们使用Unity的内置函数ObjSpaceLightDir和ObjSpaceViewDir来得到模型空间下的光照和视角方向，再利用变换矩阵rotation把它们从模型空间变换到切线空间中。

（10）由于我们在顶点着色器中完成了大部分工作，因此片元着色器中只需要采样得到切线空间下的法线方向，再在切线空间下进行光照计算即可：

```

fixed4 frag(v2f i) : SV_Target {
    fixed3 tangentLightDir = normalize(i.lightDir);
    fixed3 tangentViewDir = normalize(i.viewDir);

    // Get the texel in the normal map

```

```

    fixed4 packedNormal = tex2D(_BumpMap, i.uv.zw);
    fixed3 tangentNormal;
    // If the texture is not marked as "Normal map"
    // tangentNormal.xy = (packedNormal.xy * 2 - 1) * _BumpScale;
    // tangentNormal.z = sqrt(1.0 - saturate(dot(tangentNormal.xy,
    tangentNormal.xy)));

    // Or mark the texture as "Normal map", and use the built-in
    function
    tangentNormal = UnpackNormal(packedNormal);
    tangentNormal.xy *= _BumpScale;
    tangentNormal.z = sqrt(1.0 - saturate(dot(tangentNormal.xy,
    tangentNormal.xy)));

    fixed3 albedo = tex2D(_MainTex, i.uv).rgb * _Color.rgb;

    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;

    fixed3 diffuse = _LightColor0.rgb * albedo * max(0,
    dot(tangentNormal, tangentLightDir));

    fixed3 halfDir = normalize(tangentLightDir + tangentViewDir);
    fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0,
    dot(tangentNormal, halfDir)), _Gloss);

    return fixed4(ambient + diffuse + specular, 1.0);
}

```

在上面的代码中，我们首先利用tex2D对法线纹理\_BumpMap进行采样。正如本节一开头所讲的，法线纹理中存储的是把法线经过映射后得到的像素值，因此我们需要把它们反映射回来。如果我们没有在Unity里把该法线纹理的类型设置成Normal map（详见7.2.4节），就需要在代码中手动进行这个过程。我们首先把packedNormal的xy分量按之前提到的公式映射回法线方向，然后乘以\_BumpScale（控制凹凸程度）来得到tangentNormal的xy分量。由于法线都是单位矢量，因此tangentNormal.z分量可以由tangentNormal.xy计算而得。由于我们使用的是切线空间下的法线纹理，因此可以保证法线方向的z分量为正。在Unity中，为了方便Unity对法线纹理的存储进行优化，我们通常会把法线纹理的纹理类型标识成Normal map，Unity会根据平台来选择不同的

压缩方法。这时，如果我们再使用上面的方法来计算就会得到错误的结果，因为此时\_BumpMap的rgb分量并不再是切线空间下法线方向的xyz值了。在7.2.4节中，我们会具体解释。在这种情况下，我们可以使用Unity的内置函数UnpackNormal来得到正确的法线方向。

(11) 最后，我们为该Unity Shader设置合适的Fallback:

Fallback "Specular"
---------------------

保存后返回Unity中查看。在NormalMapTangentSpaceMat的面板上，我们使用本书资源中的Brick\_Diffuse.jpg和Brick\_Normal.jpg纹理对其赋值。我们可以调整材质面板中的Bump Scale属性来改变模型的凹凸程度。图7.15给出了不同的Bump Scale属性值下得到的结果。



▲图7.15 使用Bump Scale属性来调整模型的凹凸程度

## 2. 在世界空间下计算

现在，我们来实现第二种方法，即在世界空间下计算光照模型。我们需要在片元着色器中把法线方向从切线空间变换到世界空间下。这种方法的基本思想是：在顶点着色器中计算从切线空间到世界空间

的变换矩阵，并把它传递给片元着色器。变换矩阵的计算可以由顶点的切线、副切线和法线在世界空间下的表示来得到。最后，我们只需要在片元着色器中把法线纹理中的法线方向从切线空间变换到世界空间下即可。尽管这种方法需要更多的计算，但在需要使用Cubemap进行环境映射等情况下，我们就需要使用这种方法。

为此，我们进行如下准备工作。

(1) 使用上一节中使用的场景。

(2) 新建一个材质。在本书资源中，该材质名为NormalMapWorldSpaceMat。

(3) 新建一个Unity Shader。在本书资源中，该Shader名为Chapter7-NormalMapWorldSpace。把新的Shader赋给第2步中创建的材质。

(4) 把第2步中创建的材质赋给胶囊体。

打开Chapter7-NormalMapWorldSpace，把上一节中的代码粘贴进去，并进行如下修改：

(1) 我们需要修改顶点着色器的输出结构体v2f，使它包含从切线空间到世界空间的变换矩阵：

```
struct v2f {
    float4 pos : SV_POSITION;
    float4 uv : TEXCOORD0;
    float4 TtoW0 : TEXCOORD1;
    float4 TtoW1 : TEXCOORD2;
    float4 TtoW2 : TEXCOORD3;
};
```

我们在3.3.2节中讲到，一个插值寄存器最多只能存储float4大小的变量，对于矩阵这样的变量，我们可以把它们按行拆成多个变量再进行存储。上面代码中的TtoW0、TtoW1和TtoW2就依次存储了从切线空间到世界空间的变换矩阵的每一行。实际上，对方向矢量的变换只需要使用3×3大小的矩阵，也就是说，每一行只需要使用float3类型的变量即可。但为了充分利用插值寄存器的存储空间，我们把世界空间下的顶点位置存储在这些变量的w分量中。

(2) 修改顶点着色器，计算从切线空间到世界空间的变换矩阵：

```
v2f vert(a2v v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.uv.xy = v.texcoord.xy * _MainTex_ST.xy + _MainTex_ST.zw;
    o.uv.zw = v.texcoord.xy * _BumpMap_ST.xy + _BumpMap_ST.zw;

    float3 worldPos = mul(_Object2World, v.vertex).xyz;
    fixed3 worldNormal = UnityObjectToWorldNormal(v.normal);
    fixed3 worldTangent = UnityObjectToWorldDir(v.tangent.xyz);
    fixed3 worldBinormal = cross(worldNormal, worldTangent) *
v.tangent.w;

    // Compute the matrix that transform directions from tangent
    space to world space
    // Put the world position in w component for optimization
    o.TtoW0 = float4(worldTangent.x, worldBinormal.x,
worldNormal.x, worldPos.x);
    o.TtoW1 = float4(worldTangent.y, worldBinormal.y,
worldNormal.y, worldPos.y);
    o.TtoW2 = float4(worldTangent.z, worldBinormal.z,
worldNormal.z, worldPos.z);

    return o;
}
```

在上面的代码中，我们计算了世界空间下的顶点切线、副切线和法线的矢量表示，并把它们**按列**摆放得到从切线空间到世界空间的变换矩阵。我们把该矩阵的每一行分别存储在TtoW0、TtoW1和TtoW2

中，并把世界空间下的顶点位置的xyz分量分别存储在了这些变量的w分量中，以便充分利用插值寄存器的存储空间。

(3) 修改片元着色器，在世界空间下进行光照计算：

```
fixed4 frag(v2f i) : SV_Target {
    // Get the position in world space
    float3 worldPos = float3(i.TtoW0.w, i.TtoW1.w, i.TtoW2.w);
    // Compute the light and view dir in world space
    fixed3 lightDir = normalize(UnityWorldSpaceLightDir(worldPos));
    fixed3 viewDir = normalize(UnityWorldSpaceViewDir(worldPos));

    // Get the normal in tangent space
    fixed3 bump = UnpackNormal(tex2D(_BumpMap, i.uv.zw));
    bump.xy *= _BumpScale;
    bump.z = sqrt(1.0 - saturate(dot(bump.xy, bump.xy)));
    // Transform the normal from tangent space to world space
    bump = normalize(half3(dot(i.TtoW0.xyz, bump), dot(i.TtoW1.xyz,
    bump), dot(i.TtoW2.xyz, bump)));

    ...
}
```

我们首先从TtoW0、TtoW1和TtoW2的w分量中构建世界空间下的坐标。然后，使用内置的UnityWorldSpaceLightDir和UnityWorldSpaceViewDir函数得到世界空间下的光照和视角方向。接着，我们使用内置的UnpackNormal函数对法线纹理进行采样和解码（需要把法线纹理的格式标识成Normal map），并使用\_BumpScale对其进行缩放。最后，我们使用TtoW0、TtoW1和TtoW2存储的变换矩阵把法线变换到世界空间下。这是通过使用点乘操作来实现矩阵的每一行和法线相乘来得到的。

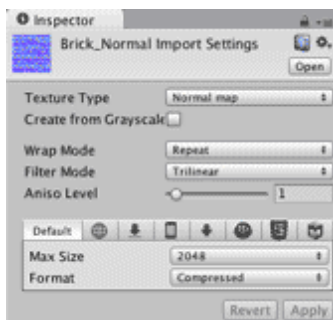
从视觉表现上，在切线空间下和在世界空间下计算光照几乎没有任何差别。在Unity 4.x版本中，在不需要使用Cubemap进行环境映射的情况下，内置的Unity Shader使用的是切线空间来进行法线映射和光照



计算。而在Unity 5.x中，所有内置的Unity Shader都使用了世界空间来进行光照计算。这也是为什么Unity 5.x中表面着色器更容易报错，因为它们使用了更多的插值寄存器来存储变换矩阵（还有一些额外的插值寄存器是用来辅助计算雾效的，更多内容可以参见19.2节）。

## 7.2.4 Unity中的法线纹理类型

上面我们提到了当把法线纹理的纹理类型标识成Normal map时，可以使用Unity的内置函数UnpackNormal来得到正确的法线方向，如图7.16所示。



▲ 图7.16 当使用UnpackNormal函数计算法线纹理中的法线方向时，需要把纹理类型标识为Normal map

当我们需要使用那些包含了法线映射的内置的Unity Shader时，必须把使用的法线纹理按上面的方式标识成Normal map才能得到正确结果（即便你忘了这么做，Unity也会在材质面板中提醒你修正这个问题），这是因为这些Unity Shader都使用了内置的UnpackNormal函数来采样法线方向。那么，当我们把纹理类型设置成Normal map时到底发生了什么？为什么要这么做呢？

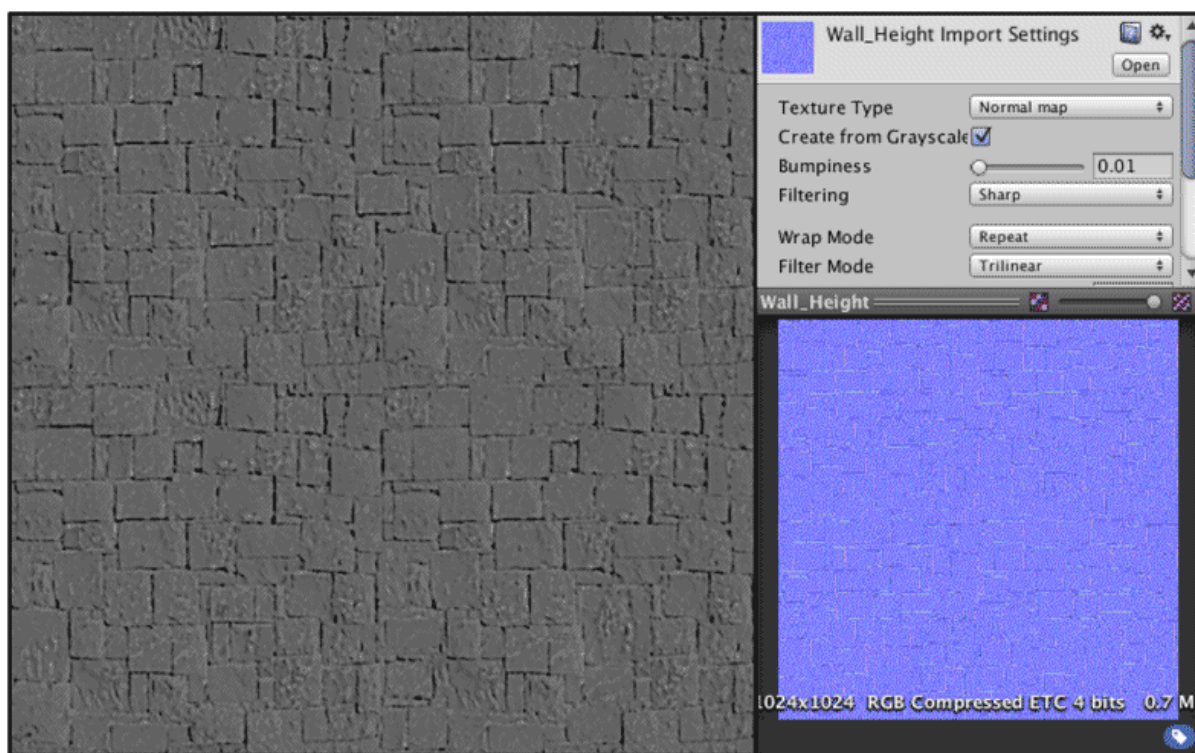
简单来说，这么做可以让Unity根据不同平台对纹理进行压缩（例如使用DXT5nm格式，具体的压缩细节可以参考：[http://tech-artists.org/wiki/Normal\\_map\\_compression](http://tech-artists.org/wiki/Normal_map_compression)），再通过UnpackNormal函数来针对不同的压缩格式对法线纹理进行正确的采样。我们可以在UnityCG.cginc里找到UnpackNormal函数的内部实现：

```
inline fixed3 UnpackNormalDXT5nm (fixed4 packednormal)
{
    fixed3 normal;
    normal.xy = packednormal.wy * 2 - 1;
    normal.z = sqrt(1 - saturate(dot(normal.xy, normal.xy)));
    return normal;
}

inline fixed3 UnpackNormal(fixed4 packednormal)
{
    #if defined(UNITY_NO_DXT5nm)
        return packednormal.xyz * 2 - 1;
    #else
        return UnpackNormalDXT5nm(packednormal);
    #endif
}
```

从代码中可以看出，在某些平台上由于使用了DXT5nm的压缩格式，因此需要针对这种格式对法线进行解码。在DXT5nm格式的法线纹理中，纹素的a通道（即w分量）对应了法线的x分量，g通道对应了法线的y分量，而纹理的r和b通道则会被舍弃，法线的z分量可以由xy分量推导而得。为什么之前的普通纹理不能按这种方式压缩，而法线就需要使用DXT5nm格式来进行压缩呢？这是因为，按我们之前的处理方式，法线纹理被当成一个和普通纹理无异的图，但实际上，它只有两个通道是真正必不可少的，因为第三个通道的值可以用另外两个推导出来（法线是单位向量，并且切线空间下的法线方向的z分量始终为正）。使用这种压缩方法就可以减少法线纹理占用的内存空间。

当我们把纹理类型设置成Normal map后，还有一个复选框是*Create from Grayscale*，那么它是做什么用的呢？读者应该还记得在本节开始我们提到过另一种凹凸映射的方法，即使用高度图，而这个复选框就是用于从高度图中生成法线纹理的。高度图本身记录的是相对高度，是一张灰度图，白色表示相对更高，黑色表示相对更低。当我们把一张高度图导入Unity后，除了需要把它的纹理类型设置成Normal map外，还需要勾选*Create from Grayscale*，这样就可以得到类似图7.17中的结果。然后，我们就可以把它和切线空间下的法线纹理同等对待了。



▲ 图7.17 当勾选了*Create from Grayscale*后，Unity会根据高度图来生成一张切线空间下的法线纹理

当勾选了*Create from Grayscale*后，还多出了两个选项——*Bumpiness*和*Filtering*。其中*Bumpiness*用于控制凹凸程度，而*Filtering*决

定我们使用哪种方式来计算凹凸程度，它有两种选项：一种是 *Smooth*，这使得生成后的法线纹理会比较平滑；另一种是 *Sharp*，它会使用 Sobel 滤波（一种边缘检测时使用的滤波器）来生成法线。Sobel 滤波的实现非常简单，我们只需要在一个  $3 \times 3$  的滤波器中计算  $x$  和  $y$  方向上的导数，然后从中得到法线即可。具体方法是：对于高度图中的每个像素，我们考虑它与水平方向和竖直方向上的像素差，把它们的差当成该点对应的法线在  $x$  和  $y$  方向上的位移，然后使用之前提到的映射函数存储成法线纹理的  $r$  和  $g$  分量即可。

## 7.3 渐变纹理

尽管在一开始，我们在渲染中使用纹理是为了定义一个物体的颜色，但后来人们发现，纹理其实可以用于存储任何表面属性。一种常见的用法就是使用渐变纹理来控制漫反射光照的结果。在之前计算漫反射光照时，我们都是使用表面法线和光照方向的点积结果与材质的反射率相乘来得到表面的漫反射光照。但有时，我们需要更加灵活地控制光照结果。这种技术在游戏《军团要塞2》（英文名：《Team Fortress 2》）中流行起来，它也是由 Valve 公司（提出半兰伯特光照技术的公司）提出来的，他们使用这种技术来渲染游戏中具有插画风格的角色。Valve 发表了一篇著名的论文来专门讲述在制作《军团要塞2》时使用的技术。

这种技术最初由 Gooch 等人在 1998 年他们发表的一篇著名的论文《A Non-Photorealistic Lighting Model For Automatic Technical Illustration》中被提出，在这篇论文中，作者提出了一种基于冷到暖色调（cool-to-warm tones）的着色技术，用来得到一种插画风格的渲染

效果。使用这种技术，可以保证物体的轮廓线相比于之前使用的传统漫反射光照更加明显，而且能够提供多种色调变化。而现在，很多卡通风格的渲染中都使用了这种技术。我们在14.1节中会专门学习如何编写一个卡通风格的Unity Shader。

在本节中，我们将学习如何使用一张渐变纹理来控制漫反射光照。在学习完本节后，我们可以得到类似图7.18中的效果。



▲图7.18 使用不同的渐变纹理控制漫反射光照，左下角给出了每张图使用的渐变纹理

可以看出，使用这种方式可以自由地控制物体的漫反射光照。不同的渐变纹理有不同的特性。例如，在左边的图中，我们使用一张从紫色调到浅黄色调的渐变纹理；而中间的图使用的渐变纹理则和《军团要塞2》中渲染人物使用的渐变纹理是类似的，它们都是从黑色逐渐向浅灰色靠拢，而且中间的分界线部分微微发红，这是因为画家在插画中往往会在阴影处使用这样的色调；右侧的渐变纹理则通常被用于卡通风格的渲染，这种渐变纹理中的色调通常是突变的，即没有平滑过渡，以此来模拟卡通中的阴影色块。

为了实现上述效果，我们需要进行如下准备工作。

(1) 在Unity中新建一个场景。在本书资源中，该场景名为Scene\_7\_3。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window -> Lighting -> Skybox中去掉场景中的天空盒子。

(2) 新建一个材质。在本书资源中，该材质名为RampTextureMat。

(3) 新建一个Unity Shader。在本书资源中，该Unity Shader名为Chapter7-RampTexture。把新的Unity Shader赋给第2步中创建的材质。

(4) 向场景中拖曳一个Suzanne模型，并把第2步中的材质赋给该模型。

(5) 保存场景。

打开新建的Chapter7-RampTexture，删除所有已有代码，并进行如下修改。

(1) 首先，我们需要为这个Shader起一个名字：

```
Shader "Unity Shaders Book/Chapter 7/Ramp Texture" {
```

(2) 我们在Properties语义块中声明一个纹理属性来存储渐变纹理：

```
Properties {  
    _Color ("Color Tint", Color) = (1,1,1,1)  
    _RampTex ("Ramp Tex", 2D) = "white" {}  
    _Specular ("Specular", Color) = (1, 1, 1, 1)  
    _Gloss ("Gloss", Range(8.0, 256)) = 20  
}
```

(3) 然后，我们在SubShader语义块中定义了一个Pass语义块，并在Pass的第一行指明了该Pass的光照模式：

```
SubShader {  
    Pass {  
        Tags { "LightMode"="ForwardBase" }  
    }  
}
```

LightMode标签是Pass标签中的一种，它用于定义该Pass在Unity的光照流水线中的角色。

(4) 然后，我们使用CGPROGRAM和ENDCG来包围住Cg代码片，以定义最重要的顶点着色器和片元着色器代码。我们使用#pragma指令来告诉Unity，我们定义的顶点着色器和片元着色器叫什么名字。在本例中，它们的名字分别是vert和frag：

```
CGPROGRAM  
  
#pragma vertex vert  
#pragma fragment frag
```

(5) 为了使用Unity内置的一些变量，如\_LightColor0，还需要包含进Unity的内置文件Lighting.cginc：

```
#include "Lighting.cginc"
```

(6) 随后，我们需要定义和Properties中各个属性类型相匹配的变量：

```
fixed4 _Color;  
sampler2D _RampTex;  
float4 _RampTex_ST;  
fixed4 _Specular;  
float _Gloss;
```



我们为渐变纹理\_RampTex定义了它的纹理属性变量\_RampTex\_ST。

(7) 定义顶点着色器的输入和输出结构体:

```
struct a2v {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
};

struct v2f {
    float4 pos : SV_POSITION;
    float3 worldNormal : TEXCOORD0;
    float3 worldPos : TEXCOORD1;
    float2 uv : TEXCOORD2;
};
```

(8) 定义顶点着色器:

```
v2f vert(a2v v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.worldNormal = UnityObjectToWorldNormal(v.normal);

    o.worldPos = mul(_Object2World, v.vertex).xyz;

    o.uv = TRANSFORM_TEX(v.texcoord, _RampTex);

    return o;
}
```

我们使用了内置的TRANSFORM\_TEX宏来计算经过平铺和偏移后的纹理坐标。

(9) 接下来是关键片元着色器:

```
fixed4 frag(v2f i) : SV_Target {
    fixed3 worldNormal = normalize(i.worldNormal);
    fixed3 worldLightDir =
```

```

normalize(UnityWorldSpaceLightDir(i.worldPos));

    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;

    // Use the texture to sample the diffuse color
    fixed halfLambert = 0.5 * dot(worldNormal, worldLightDir) +
0.5;
    fixed3 diffuseColor = tex2D(_RampTex, fixed2(halfLambert,
halfLambert)).rgb * _Color.rgb;

    fixed3 diffuse = _LightColor0.rgb * diffuseColor;

    fixed3 viewDir = normalize(UnityWorldSpaceViewDir(i.worldPos));
    fixed3 halfDir = normalize(worldLightDir + viewDir);
    fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0,
dot(worldNormal, halfDir)), _Gloss);

    return fixed4(ambient + diffuse + specular, 1.0);
}

```

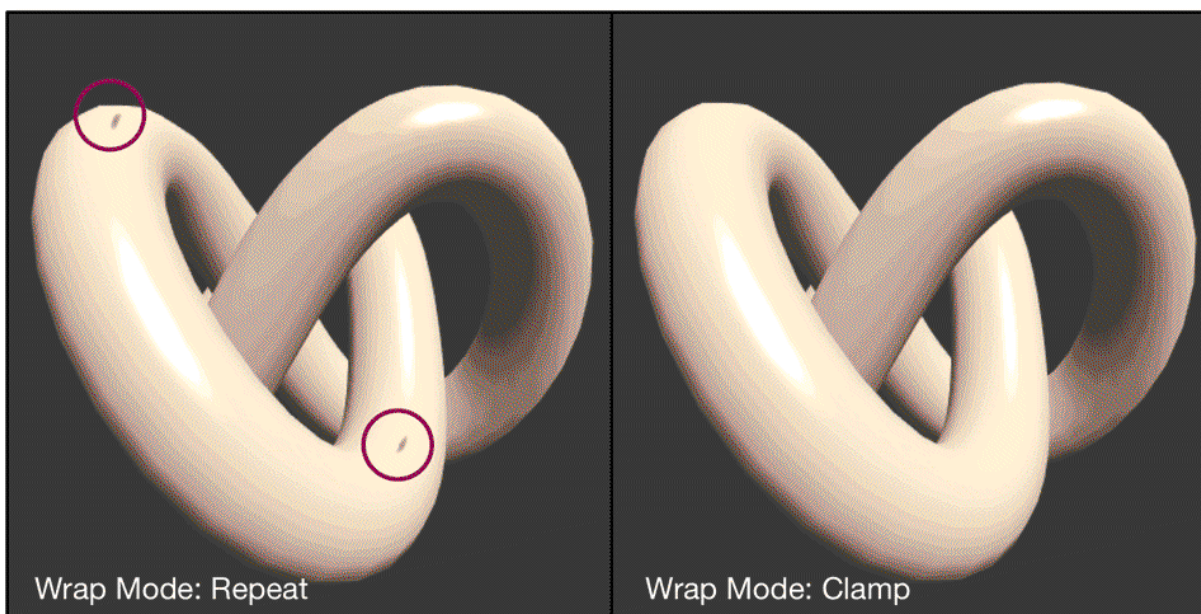
在上面的代码中，我们使用6.4.3节中提到的半兰伯特模型，通过对法线方向和光照方向的点积做一次0.5倍的缩放以及一个0.5大小的偏移来计算半兰伯特部分`halfLambert`。这样，我们得到的`halfLambert`的范围被映射到了[0, 1]之间。之后，我们使用`halfLambert`来构建一个纹理坐标，并用这个纹理坐标对渐变纹理`_RampTex`进行采样。由于`_RampTex`实际就是一个一维纹理（它在纵轴方向上颜色不变），因此纹理坐标的`u`和`v`方向我们都使用了`halfLambert`。然后，把从渐变纹理采样得到的颜色和材质颜色`_Color`相乘，得到最终的漫反射颜色。剩下的代码就是计算高光反射和环境光，并把它们的结果进行相加。相信读者已经对这些步骤非常熟悉了。

（10）最后，我们为该Unity Shader设置合适的Fallback:

```
Fallback "Specular"
```

保存后返回场景。我们在本书资源中提供了多种渐变纹理，如 Ramp\_Texture0.psd和Ramp\_Texture1.psd等。读者可以尝试把不同的渐变纹理拖曳到材质面板查看效果。

**需要注意的是**，我们需要把渐变纹理的Wrap Mode设为Clamp模式，以防止对纹理进行采样时由于浮点数精度而造成的问题。图7.19给出了Wrap Mode分别为Repeat和Clamp模式的效果对比。



▲ 图7.19 Wrap Mode分别为Repeat和Clamp模式的效果对比

可以看出，左图（使用Repeat模式）中在高光区域有一些黑点。这是由浮点精度造成的，当我们使用fixed2(halfLambert, halfLambert)对渐变纹理进行采样时，虽然理论上halfLambert的值在[0, 1]之间，但可能会有1.000 01这样的值出现。如果我们使用的是Repeat模式，此时就会舍弃整数部分，只保留小数部分，得到的值就是0.000 01，对应了渐变图中最左边的值，即黑色。因此，就会出现图中这样在高光区域反而

有黑点的情况。我们只需要把渐变纹理的Wrap Mode设为Clamp模式就可以解决这种问题。

## 7.4 遮罩纹理

**遮罩纹理 (mask texture)** 是本章要介绍的最后一种纹理，它非常有用，在很多商业游戏中都可以见到它的身影。那么什么是遮罩呢？简单来讲，遮罩允许我们可以保护某些区域，使它们免于某些修改。例如，在之前的实现中，我们都是把高光反射应用到模型表面的所有地方，即所有的像素都使用同样大小的高光强度和高光指数。但有时，我们希望模型表面某些区域的反光强烈一些，而某些区域弱一些。为了得到更加细腻的效果，我们就可以使用一张遮罩纹理来控制光照。另一种常见的应用是在制作地形材质时需要混合多张图片，例如表现草地的纹理、表现石子的纹理、表现裸露土地的纹理等，使用遮罩纹理可以控制如何混合这些纹理。

使用遮罩纹理的流程一般是：通过采样得到遮罩纹理的纹素值，然后使用其中某个（或某几个）通道的值（例如`texel.r`）来与某种表面属性进行相乘，这样，当该通道的值为0时，可以保护表面不受该属性的影响。总而言之，使用遮罩纹理可以让美术人员更加精准（像素级别）地控制模型表面的各种性质。

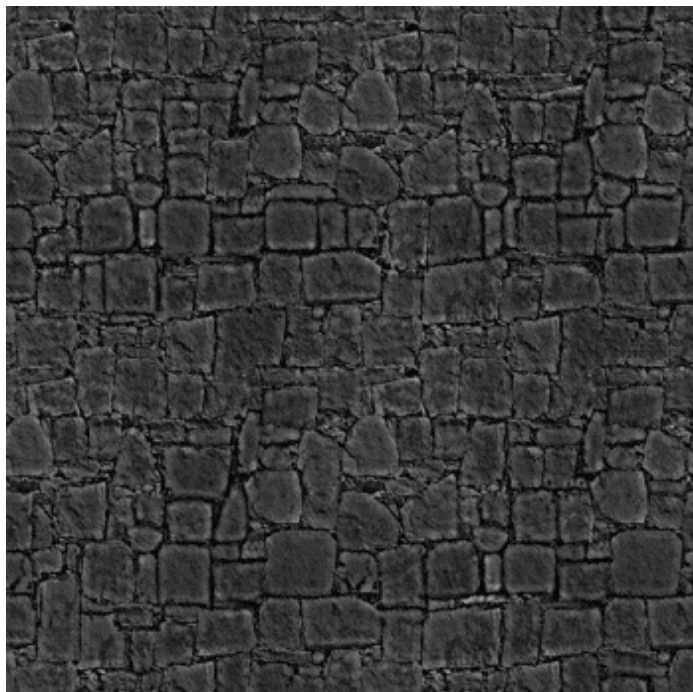
### 7.4.1 实践

在本节中，我们将学习如何使用一张高光遮罩纹理，逐像素地控制模型表面的高光反射强度。图17.20显示了只包含漫反射、未使用遮罩的高光反射和使用遮罩的高光反射的对比效果。



▲图7.20 使用高光遮罩纹理。从左到右：只包含漫反射，未使用遮罩的高光反射，使用遮罩的高光反射

我们使用的遮罩纹理如图7.21所示。可以看出，遮罩纹理可以让我们更加精细地控制光照细节，得到更细腻的效果。



▲图7.21 本节使用的高光遮罩纹理

为了在Unity Shader中实现上述效果，我们需要进行如下准备工作。

(1) 在Unity中新建一个场景。在本书资源中，该场景名为Scene\_7\_4。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window -> Lighting -> Skybox中去掉场景中的天空盒子。

(2) 新建一个材质。在本书资源中，该材质名为MaskTextureMat。

(3) 新建一个Unity Shader。在本书资源中，该Unity Shader名为Chapter7-MaskTexture。把新的Unity Shader赋给第2步中创建的材质。

(4) 在场景中创建一个胶囊体，并把第2步中的材质赋给该胶囊体。

(5) 保存场景。

打开新建的Chapter7-MaskTexture，删除所有已有代码，并进行如下修改：

(1) 首先，我们需要为这个Shader起一个名字：

```
Shader "Unity Shaders Book/Chapter 7/Mask Texture" {
```

(2) 我们需要在Properties语义块中声明更多的变量来控制高光反射：

```
Properties {  
    _Color ("Color Tint", Color) = (1,1,1,1)  
    _MainTex ("Main Tex", 2D) = "white" {}  
    _BumpMap ("Normal Map", 2D) = "bump" {}  
    _BumpScale("Bump Scale", Float) = 1.0  
    _SpecularMask ("Specular Mask", 2D) = "white" {}  
    _SpecularScale ("Specular Scale", Float) = 1.0  
    _Specular ("Specular", Color) = (1, 1, 1, 1)  
    _Gloss ("Gloss", Range(8.0, 256)) = 20  
}
```

上面属性中的\_SpecularMask即是我们需要使用的高光反射遮罩纹理，\_SpecularScale则是用于控制遮罩影响度的系数。

(3) 然后，我们在SubShader语义块中定义了一个Pass语义块，并在Pass的第一行指明了该Pass的光照模式：

```
SubShader {  
    Pass {  
        Tags { "LightMode"="ForwardBase" }  
    }  
}
```

LightMode标签是Pass标签中的一种，它用于定义该Pass在Unity的光照流水线中的角色。

(4) 然后，我们使用CGPROGRAM和ENDCG来包围住Cg代码片，以定义最重要的顶点着色器和片元着色器代码。我们使用#pragma指令来告诉Unity，我们定义的顶点着色器和片元着色器叫什么名字。在本例中，它们的名字分别是vert和frag：

```
CGPROGRAM  
  
#pragma vertex vert  
#pragma fragment frag
```

(5) 为了使用Unity内置的一些变量，如\_LightColor0，还需要包含进Unity的内置文件Lighting.cginc：



```
#include "Lighting.cginc"
```

(6) 随后，我们需要定义和Properties中各个属性类型相匹配的变量：

```
fixed4 _Color;  
sampler2D _MainTex;  
float4 _MainTex_ST;  
sampler2D _BumpMap;  
float _BumpScale;  
sampler2D _SpecularMask;  
float _SpecularScale;  
fixed4 _Specular;  
float _Gloss;
```

我们为主纹理\_MainTex、法线纹理\_BumpMap和遮罩纹理\_SpecularMask定义了它们共同使用的纹理属性变量\_MainTex\_ST。这意味着，在材质面板中修改主纹理的平铺系数和偏移系数会同时影响3个纹理的采样。使用这种方式可以让我们节省需要存储的纹理坐标数目，如果我们为每一个纹理都使用一个单独的属性变量TextureName\_ST，那么随着使用的纹理数目的增加，我们会迅速占满顶点着色器中可以使用的插值寄存器。而很多时候，我们不需要对纹理进行平铺和位移操作，或者很多纹理可以使用同一种平铺和位移操作，此时我们就可以对这些纹理使用同一个变换后的纹理坐标进行采样。

(7) 定义顶点着色器的输入和输出结构体：

```
struct a2v {  
    float4 vertex : POSITION;  
    float3 normal : NORMAL;  
    float4 tangent : TANGENT;  
    float4 texcoord : TEXCOORD0;  
};
```

```

struct v2f {
    float4 pos : SV_POSITION;
    float2 uv : TEXCOORD0;
    float3 lightDir: TEXCOORD1;
    float3 viewDir : TEXCOORD2;
};

```

(8) 在顶点着色器中，我们对光照方向和视角方向进行了坐标空间的变换，把它们从模型空间变换到了切线空间中，以便在片元着色器中和法线进行光照运算：

```

v2f vert(a2v v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.uv.xy = v.texcoord.xy * _MainTex_ST.xy + _MainTex_ST.zw;

    TANGENT_SPACE_ROTATION;
    o.lightDir = mul(rotation, ObjSpaceLightDir(v.vertex)).xyz;
    o.viewDir = mul(rotation, ObjSpaceViewDir(v.vertex)).xyz;

    return o;
}

```

(9) 使用遮罩纹理的地方是片元着色器。我们使用它来控制模型表面的高光反射强度：

```

fixed4 frag(v2f i) : SV_Target {
    fixed3 tangentLightDir = normalize(i.lightDir);
    fixed3 tangentViewDir = normalize(i.viewDir);

    fixed3 tangentNormal = UnpackNormal(tex2D(_BumpMap, i.uv));
    tangentNormal.xy *= _BumpScale;
    tangentNormal.z = sqrt(1.0 - saturate(dot(tangentNormal.xy,
tangentNormal.xy)));

    fixed3 albedo = tex2D(_MainTex, i.uv).rgb * _Color.rgb;

    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;

    fixed3 diffuse = _LightColor0.rgb * albedo * max(0,
dot(tangentNormal, tangentLightDir));

    fixed3 halfDir = normalize(tangentLightDir + tangentViewDir);
}

```

```
// Get the mask value
fixed specularMask = tex2D(_SpecularMask, i.uv).r *
_SpecularScale;
// Compute specular term with the specular mask
fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0,
dot(tangentNormal, halfDir)), _Gloss) * specularMask;

return fixed4(ambient + diffuse + specular, 1.0);
}
```

环境光照和漫反射光照和之前使用过的代码完全一样。在计算高光反射时，我们首先对遮罩纹理`_SpecularMask`进行采样。由于本书使用的遮罩纹理中每个纹素的`rgb`分量其实都是一样的，表明了该点对应的高光反射强度，在这里我们选择使用`r`分量来计算掩码值。然后，我们用得到的掩码值和`_SpecularScale`相乘，一起来控制高光反射的强度。

**需要说明的是**，我们使用的这张遮罩纹理其实有很多空间被浪费了——它的`rgb`分量存储的都是同一个值。在实际的游戏制作中，我们往往会充分利用遮罩纹理中的每一个颜色通道来存储不同的表面属性，我们会在7.4.2节中介绍这样一个例子。

(10) 最后，我们为该Unity Shader设置了合适的Fallback:

```
Fallback "Specular"
```

## 7.4.2 其他遮罩纹理

在真实的游戏制作过程中，遮罩纹理已经不止限于保护某些区域使它们免于某些修改，而是可以存储任何我们希望逐像素控制的表面属性。通常，我们会充分利用一张纹理的`RGBA`四个通道，用于存储不同的属性。例如，我们可以把高光反射的强度存储在`R`通道，把边缘光

照的强度存储在G通道，把高光反射的指数部分存储在B通道，最后把自发光强度存储在A通道。

在游戏《DOTA 2》的开发中，开发人员为每个模型使用了4张纹理：一张用于定义模型颜色，一张用于定义表面法线，另外两张则都是遮罩纹理。这样，两张遮罩纹理提供了共8种额外的表面属性，这使得游戏中的人物材质自由度很强，可以支持很多高级的模型属性。读者可以在他们的官网上找到关于《DOTA 2》的更加详细的制作资料，包括游戏中的人物模型、纹理以及制作手册等。这是非常好的学习资料。

## 第8章 透明效果

透明是游戏中经常要使用的一种效果。在实时渲染中要实现透明效果，通常会在渲染模型时控制它的**透明通道（Alpha Channel）**。当开启透明混合后，当一个物体被渲染到屏幕上时，每个片元除了颜色值和深度值之外，它还有另一个属性——透明度。当透明度为1时，表示该像素是完全不透明的，而当其为0时，则表示该像素完全不会显示。

在Unity中，我们通常使用两种方法来实现透明效果：第一种是使用**透明度测试（Alpha Test）**，这种方法其实无法得到真正的半透明效果；另一种是**透明度混合（Alpha Blending）**。

在之前的学习中，我们从没有强调过渲染顺序的问题。也就是说，当场景中包含很多模型时，我们并没有考虑是先渲染A，再渲染B，最后再渲染C，还是按照其他的顺序来渲染。事实上，对于不透明（opaque）物体，不考虑它们的渲染顺序也能得到正确的排序效果，这是由于强大的深度缓冲（depth buffer，也被称为z-buffer）的存在。在实时渲染中，深度缓冲是用于解决可见性（visibility）问题的，它可以决定哪个物体的哪些部分会被渲染在前面，而哪些部分会被其他物体遮挡。它的基本思想是：根据深度缓存中的值来判断该片元距离摄像机的距离，当渲染一个片元时，需要把它的深度值和已经存在于深度缓冲中的值进行比较（如果开启了深度测试），如果它的值距离摄像机更远，那么说明这个片元不应该被渲染到屏幕上（有物体挡住了

它)；否则，这个片元应该覆盖掉此时颜色缓冲中的像素值，并把它深度值更新到深度缓冲中（如果开启了深度写入）。

使用深度缓冲，可以让我们不用关心不透明物体的渲染顺序，例如A挡住B，即便我们先渲染A再渲染B也不用担心B会遮盖掉A，因为在进行深度测试时会判断出B距离摄像机更远，也就不会写入到颜色缓冲中。但如果想要实现透明效果，事情就不那么简单了，这是因为，当使用透明度混合时，我们关闭了深度写入（ZWrite）。

简单来说，透明度测试和透明度混合的基本原理如下。

- **透明度测试：**它采用一种“霸道极端”的机制，只要一个片元的透明度不满足条件（通常是小于某个阈值），那么它对应的片元就会被舍弃。被舍弃的片元将不会再进行任何处理，也不会对颜色缓冲产生任何影响；否则，就会按照普通的不透明物体的处理方式来处理它，即进行深度测试、深度写入等。也就是说，透明度测试是不需要关闭深度写入的，它和其他不透明物体最大的不同就是它会根据透明度来舍弃一些片元。虽然简单，但是它产生的效果也很极端，要么完全透明，即看不到，要么完全不透明，就像不透明物体那样。
- **透明度混合：**这种方法可以得到真正的半透明效果。它会使用当前片元的透明度作为混合因子，与已经存储在颜色缓冲中的颜色值进行混合，得到新的颜色。但是，透明度混合需要关闭深度写入（我们下面会讲为什么需要关闭），这使得我们要非常小心物体的渲染顺序。需要注意的是，透明度混合只关闭了深度写入，但没有关闭深度测试。这意味着，当使用透明度混合渲染一个片

元时，还是会比较它的深度值与当前深度缓冲中的深度值，如果它的深度值距离摄像机更远，那么就不会再进行混合操作。这一点决定了，当一个不透明物体出现在一个透明物体的前面，而我们先渲染了不透明物体，它仍然可以正常地遮挡住透明物体。也就是说，对于透明度混合来说，深度缓冲是只读的。

## 8.1 为什么渲染顺序很重要

前面说到，对于透明度混合技术，需要关闭深度写入，此时我们就需要小心处理透明物体的渲染顺序。那么，我们为什么要关闭深度写入呢？如果不关闭深度写入，一个半透明表面背后的表面本来是可以透过它被我们看到的，但由于深度测试时判断结果是该半透明表面距离摄像机更近，导致后面的表面将会被剔除，我们也就无法透过半透明表面看到后面的物体了。但是，我们由此就破坏了深度缓冲的工作机制，而这是一个**非常非常非常**（重要的事情要讲3遍）糟糕的事情，尽管我们不得不这样做。关闭深度写入导致渲染顺序将变得非常重要。

我们来考虑最简单的情况。假设场景里有两个物体A和B，如图8.1所示，其中A是半透明物体，而B是不透明物体。

我们来考虑不同的渲染顺序会有什么结果。

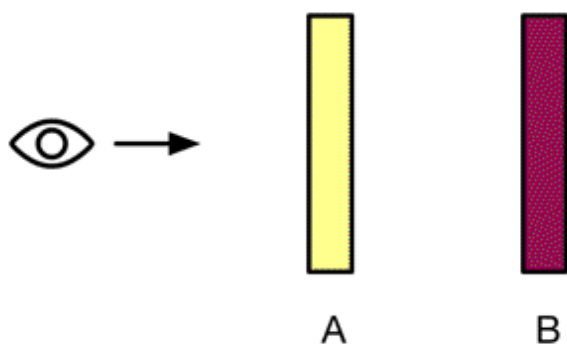
- 第一种情况，我们先渲染B，再渲染A。那么由于不透明物体开启了深度测试和深度写入，而此时深度缓冲中没有任何有效数据，因此B首先会写入颜色缓冲和深度缓冲。随后我们渲染A，透明物体仍然会进行深度测试，因此我们发现和B相比A距离摄像机更



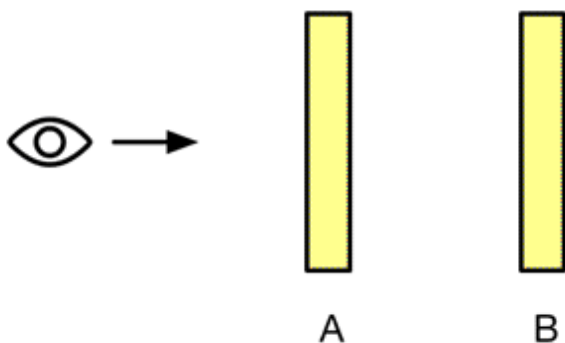
近，因此，我们会使用A的透明度来和颜色缓冲中的B的颜色进行混合，得到正确的半透明效果。

- 第二种情况，我们先渲染A，再渲染B。渲染A时，深度缓冲区中没有任何有效数据，因此A直接写入颜色缓冲，但由于对半透明物体关闭了深度写入，因此A不会修改深度缓冲。等到渲染B时，B会进行深度测试，它发现，“咦，深度缓存中还没有人来过，那我就放心地写入颜色缓冲了！”，结果就是B会直接覆盖A的颜色。从视觉上来看，B就出现在了A的前面，而这是错误的。

从这个例子可以看出，当关闭了深度写入后，渲染顺序是多么重要。由此我们知道，我们应该在不透明物体渲染完之后再渲染半透明物体。那么，如果都是半透明物体，渲染顺序还重要吗？答案是肯定的。还是假设场景里有两个物体A和B，如图8.2所示，其中A和B都是半透明物体。



▲图8.1 场景中有两个物体，其中A（黄色）是半透明物体，B（紫色）是不透明物体



▲图8.2 场景中有两个物体，其中A和B都是半透明物体

我们还是考虑不同的渲染顺序有什么不同结果。

- 第一种情况，我们先渲染B，再渲染A。那么B会正常写入颜色缓冲，然后A会和颜色缓冲中的B颜色进行混合，得到正确的半透明效果。
- 第二种情况，我们先渲染A，再渲染B。那么A会先写入颜色缓冲，随后B会和颜色缓冲中的A进行混合，这样混合结果会完全反过来，看起来就好像B在A的前面，得到的就是错误的半透明结构。

从这个例子可以看出，半透明物体之间也是要符合一定的渲染顺序的。

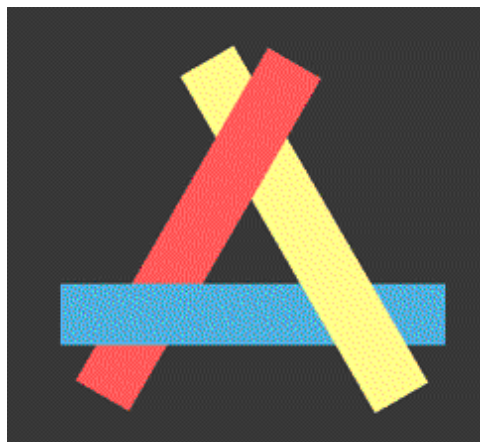
基于这两点，渲染引擎一般都会先对物体进行排序，再渲染。常用的方法是。

(1) 先渲染所有不透明物体，并开启它们的深度测试和深度写入。

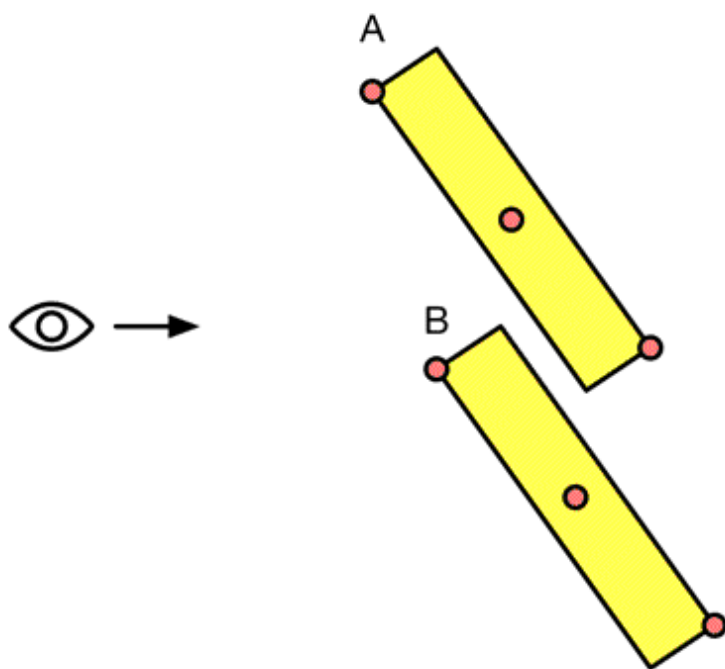
(2) 把半透明物体按它们距离摄像机的远近进行排序，然后按照从后往前的顺序渲染这些半透明物体，并开启它们的深度测试，但关闭深度写入。

那么，问题都解决了吗？不幸的是，仍然没有。在一些情况下，半透明物体还是会出现“穿帮镜头”。如果我们仔细想想的话，上面给出的第2步中渲染顺序仍然是含糊不清的——“按它们距离摄像机的远近进行排序”，那么它们距离摄像机的远近是如何决定的呢？读者可能会马上脱口而出，“就是距离摄像的深度值嘛！”但是，深度缓冲中的值其实是像素级别的，即每个像素有一个深度值，但是现在我们对单个物体级别进行排序，这意味着排序结果是，要么物体A全部在B前面渲染，要么A全部在B后面渲染。但如果存在循环重叠的情况，那么使用这种方法就永远无法得到正确的结果。图8.3给出了3个物体循环重叠的情况。

在图8.3中，由于3个物体互相重叠，我们不可能得到一个正确的排序顺序。这种时候，我们可以选择把物体拆分成两个部分，然后再进行正确的排序。但即便我们通过分割的方法解决了循环覆盖的问题，还是会有其他的情况来“捣乱”。考虑图8.4给出的情况。



▲图8.3 循环重叠的半透明物体总是无法得到正确的半透明效果



▲图8.4 使用哪个深度对物体进行排序。红色点分别标明了网格上距离摄像机最近的点、最远的点以及网格中点

这里的问题是：如何排序？我们知道，一个物体的网格结构往往占据了空间中的某一块区域，也就是说，这个网格上每一个点的深度值可能都是不一样的，我们选择哪个深度值来作为整个物体的深度值和其他物体进行排序呢？是网格中点吗？还是最远的点？还是最近的点？不幸的是，对于图8.4中的情况，选择哪个深度值都会得到错误的结果，我们的排序结果总是A在B的前面，但实际上A有一部分被B遮挡了。这也意味着，一旦选定了一种判断方式后，在某些情况下半透明物体之间一定会出现错误的遮挡问题。这种问题的解决方法通常也是分割网格。

尽管结论是，总是会有一些情况打乱我们的阵脚，但由于上述方法足够有效并且容易实现，因此大多数游戏引擎都使用了这样的方法。为了减少错误排序的情况，我们可以尽可能让模型是凸面体，并且考虑将复杂的模型拆分成可以独立排序的多个子模型等。其实就算排序错误结果有时也不会非常糟糕，如果我们不想分割网格，可以试着让透明通道更加柔和，使穿插看起来并不是那么明显。我们也可以使用开启了深度写入的半透明效果来近似模拟物体的半透明（详见8.5节）。

下面，我们就来看一下Unity是如何解决排序问题的。

## 8.2 Unity Shader的渲染顺序

Unity为了解决渲染顺序的问题提供了**渲染队列（render queue）**这一解决方案。我们可以使用SubShader的**Queue**标签来决定我们的模型将归于哪个渲染队列。Unity在内部使用一系列整数索引来表示每个渲染队列，且索引号越小表示越早被渲染。在Unity 5中，Unity提前定义了5个渲染队列（与Unity 5之前的版本相比多了一个AlphaTest渲染队列），当然在每个队列中间我们可以使用其他队列。表8.1给出了这5个提前定义的渲染队列以及它们的描述。

表8.1 Unity提前定义的5个渲染队列

名称	队列索引号	描 述

名称	队列索引号	描 述
Background	1000	这个渲染队列会在任何其他队列之前被渲染，我们通常使用该队列来渲染那些需要绘制在背景上的物体
Geometry	2000	默认的渲染队列，大多数物体都使用这个队列。不透明物体使用这个队列
AlphaTest	2450	需要透明度测试的物体使用这个队列。在Unity 5中它从Geometry队列中被单独分出来，这是因为在所有不透明物体渲染之后再渲染它们会更加高效
Transparent	3000	这个队列中的物体会在所有Geometry和AlphaTest物体渲染后，再按 <b>从后往前</b> 的顺序进行渲染。任何使用了透明度混合（例如关闭了深度写入的Shader）的物体都应该使用该队列
Overlay	4000	该队列用于实现一些叠加效果。任何需要在最后渲染的物体都应该使用该队列

因此，如果我们想要通过透明度测试实现透明效果，代码中应该包含类似下面的代码：

```
SubShader {
    Tags { "Queue"="AlphaTest" }
    Pass {
        ...
    }
}
```

如果我们想要通过透明度混合来实现透明效果，代码中应该包含类似下面的代码：

```
SubShader {  
    Tags { "Queue"="Transparent" }  
    Pass {  
        ZWrite Off  
        ...  
    }  
}
```

其中，**ZWrite Off**用于关闭深度写入，在这里我们选择把它写在Pass中。我们也可以把它写在SubShader中，这意味着该SubShader下的所有Pass都会关闭深度写入。

## 8.3 透明度测试

我们来看一下如何在Unity中实现透明度测试的效果。在上面我们已经知道了透明度测试的工作原理。

**透明度测试：** 只要一个片元的透明度不满足条件（通常是小于某个阈值），那么它对应的片元就会被舍弃。被舍弃的片元将不会再进行任何处理，也不会对颜色缓冲产生任何影响；否则，就会按照普通的不透明物体的处理方式来处理它。

通常，我们会在片元着色器中使用clip函数来进行透明度测试。clip是Cg中的一个函数，它的定义如下。

**函数：** void clip(float4 x); void clip(float3 x); void clip(float2 x); void clip(float1 x); void clip(float x);



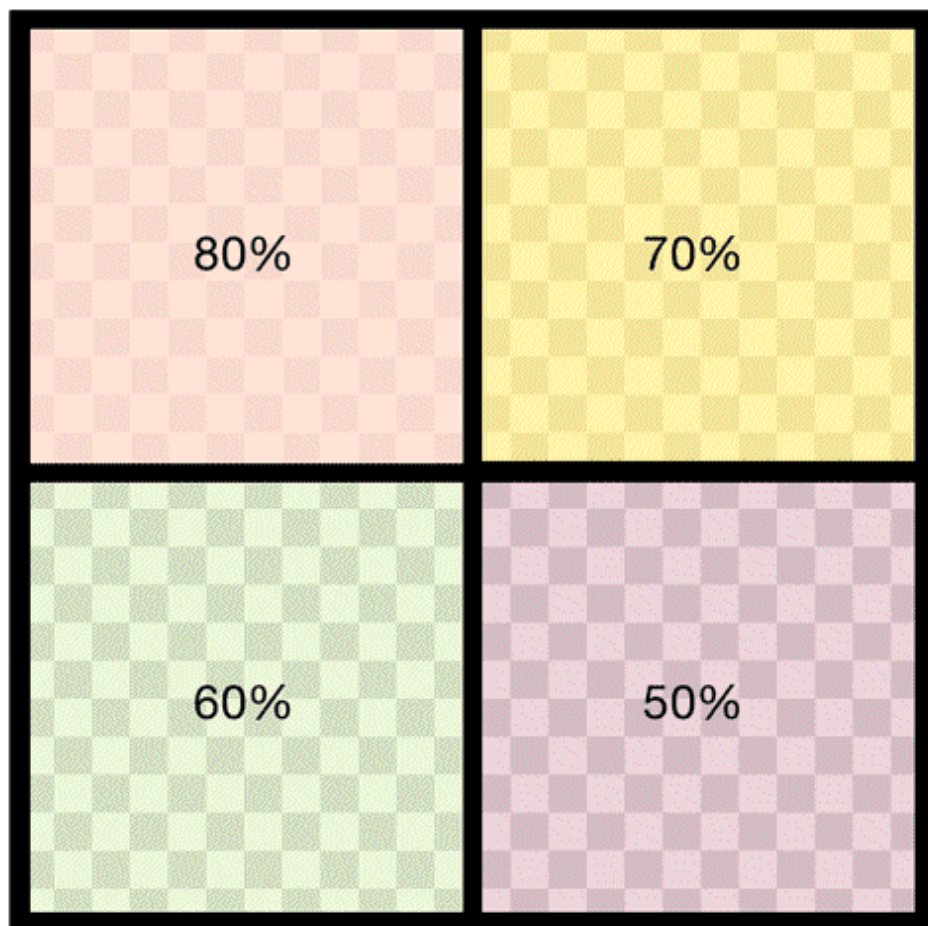
**参数：**裁剪时使用的标量或矢量条件。

**描述：**如果给定参数的任何一个分量是负数，就会舍弃当前像素的输出颜色。它等同于下面的代码：

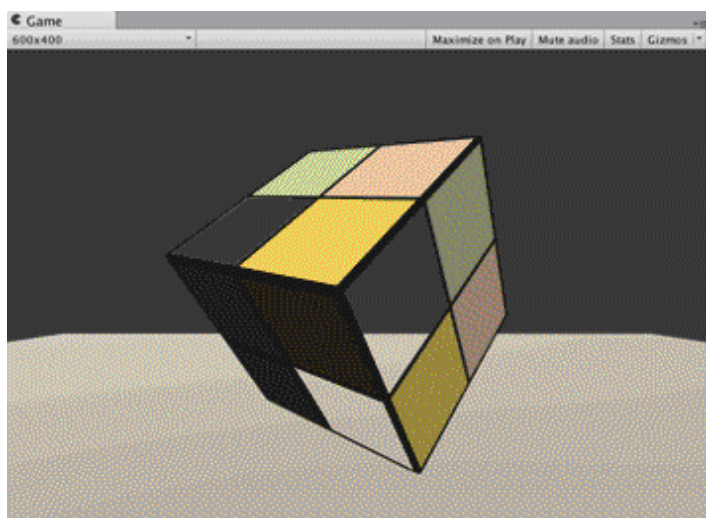
```
void clip(float4 x)
{
    if (any(x < 0))
        discard;
}
```

在本节中，我们使用图8.5中的半透明纹理来实现透明度测试。在本书资源中，该纹理名为transparent\_texture.psd。该透明纹理在不同区域的透明度也不同，我们通过它来查看透明度测试的效果。

在学习完本节后，我们可以得到类似图8.6中的效果。



▲ 图8.5 一张透明纹理，其中每个方格的透明度都不同



▲图8.6 透明度测试

为此，我们需要进行如下准备工作。

(1) 在Unity中新建一个场景。在本书资源中，该场景名为Scene\_8\_3。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window -> Lighting -> Skybox中去掉场景中的天空盒子。

(2) 新建一个材质。在本书资源中，该材质名为AlphaTestMat。

(3) 新建一个Unity Shader。在本书资源中，该Unity Shader名为Chapter8-AlphaTest。把新的Unity Shader赋给第2步中创建的材质。

(4) 在场景中创建一个立方体，并把第2步中的材质赋给该模型。创建一个平面，使得平面位于立方体下面。

(5) 保存场景。

打开新建的Chapter8-AlphaTest，删除所有已有代码，并进行如下修改。

(1) 首先，我们需要为这个Shader起一个名字：

```
Shader "Unity Shaders Book/Chapter 8/Alpha Test" {
```

(2) 为了在材质面板中控制透明度测试时使用的阈值，我们在Properties语义块中声明一个范围在[0, 1]之间的属性\_Cutoff：

```
Properties {  
    _Color ("Main Tint", Color) = (1,1,1,1)
```

```
_MainTex ("Main Tex", 2D) = "white" {}  
_Cutoff ("Alpha Cutoff", Range(0, 1)) = 0.5  
}
```

`_Cutoff`参数用于决定我们调用`clip`进行透明度测试时使用的判断条件。它的范围是`[0, 1]`，这是因为纹理像素的透明度就是在此范围内。

(3) 然后，我们在`SubShader`语义块中定义了一个`Pass`语义块：

```
SubShader {  
    Tags {"Queue"="AlphaTest" "IgnoreProjector"="True"  
"RenderTarget"="TransparentCutout"}  
  
    Pass {  
        Tags { "LightMode"="ForwardBase" }  
    }  
}
```

我们在8.2节中已经知道渲染顺序的重要性，并且知道在Unity中透明度测试使用的渲染队列是名为`AlphaTest`的队列，因此我们需要把`Queue`标签设置为`AlphaTest`。而`RenderTarget`标签可以让Unity把这个`Shader`归入到提前定义的组（这里就是`TransparentCutout`组）中，以指明该`Shader`是一个使用了透明度测试的`Shader`。`RenderTarget`标签通常被用于着色器替换功能。我们还把`IgnoreProjector`设置为`True`，这意味着这个`Shader`不会受到投影器（`Projectors`）的影响。通常，使用了透明度测试的`Shader`都应该在`SubShader`中设置这三个标签。最后，`LightMode`标签是`Pass`标签中的一种，它用于定义该`Pass`在Unity的光照流水线中的角色。只有定义了正确的`LightMode`，我们才能正确得到一些Unity的内置光照变量，例如`_LightColor0`。

(4) 然后，我们使用`CGPROGRAM`和`ENDCG`来包围住Cg代码片，来定义最重要的顶点着色器和片元着色器代码。首先，我们使用

`#pragma`指令来告诉Unity，我们定义的顶点着色器和片元着色器叫什么名字。在本例中，它们的名字分别是`vert`和`frag`：

```
CGPROGRAM

#pragma vertex vert
#pragma fragment frag
```

(5) 为了使用Unity内置的一些变量，如`_LightColor0`，还需要包含进Unity的内置文件`Lighting.cginc`：

```
#include "Lighting.cginc"
```

(6) 为了和`Properties`语义块中声明的属性建立联系，我们需要定义和各个属性类型相匹配的变量：

```
fixed4 _Color;
sampler2D _MainTex;
float4 _MainTex_ST;
fixed _Cutoff;
```

由于`_Cutoff`的范围在`[0, 1]`，因此我们可以使用`fixed`精度来存储它。

(7) 然后，我们定义了顶点着色器的输入和输出结构体，接着定义顶点着色器：

```
struct a2v {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
};

struct v2f {
    float4 pos : SV_POSITION;
    float3 worldNormal : TEXCOORD0;
    float3 worldPos : TEXCOORD1;
```

```

    float2 uv : TEXCOORD2;
};

v2f vert(a2v v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.worldNormal = UnityObjectToWorldNormal(v.normal);

    o.worldPos = mul(_Object2World, v.vertex).xyz;

    o.uv = TRANSFORM_TEX(v.texcoord, _MainTex);

    return o;
}

```

上面的代码我们已经见到过很多次了，我们在顶点着色器计算出世界空间的法线方向和顶点位置以及变换后的纹理坐标，再把它们传递给片元着色器。

(8) 最重要的透明度测试的代码在片元着色器中：

```

fixed4 frag(v2f i) : SV_Target {
    fixed3 worldNormal = normalize(i.worldNormal);
    fixed3 worldLightDir =
normalize(UnityWorldSpaceLightDir(i.worldPos));

    fixed4 texColor = tex2D(_MainTex, i.uv);

    // Alpha test
    clip (texColor.a - _Cutoff);
    // Equal to
    // if ((texColor.a - _Cutoff) < 0.0) {
    //     discard;
    // }

    fixed3 albedo = texColor.rgb * _Color.rgb;

    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;

    fixed3 diffuse = _LightColor0.rgb * albedo * max(0,
dot(worldNormal, worldLightDir));

    return fixed4(ambient + diffuse, 1.0);
}

```

前面我们已经提到过clip函数的定义，它会判断它的参数，即texColor.a - \_Cutoff是否为负数，如果是就会舍弃该片元的输出。也就是说，当texColor.a小于材质参数\_Cutoff时，该片元就会产生完全透明的效果。使用clip函数等同于先判断参数是否小于零，如果是就使用discard指令来显式剔除该片元。后面的代码和之前使用过的完全一样，我们计算得到环境光照和漫反射光照，把它们相加后再进行输出。

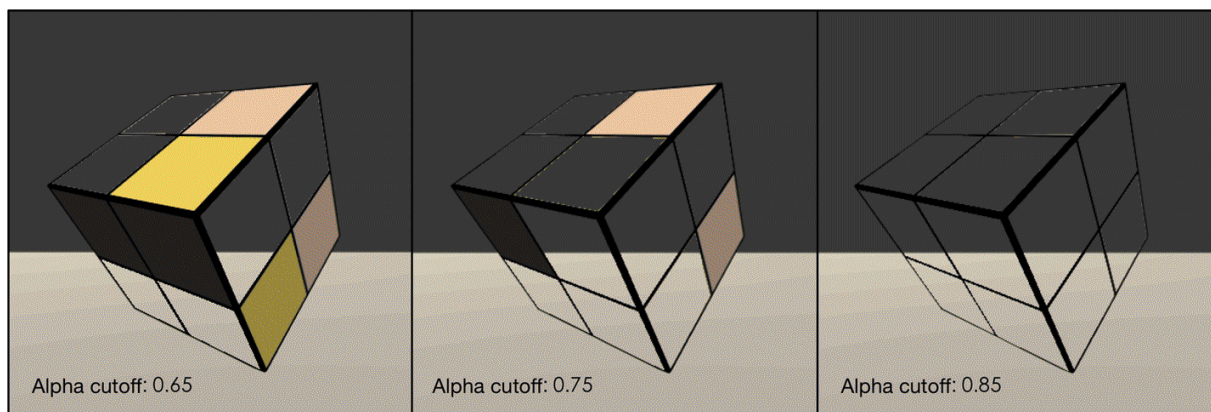
(9) 最后，我们需要为这个Unity Shader设置合适的Fallback:

Fallback "Transparent/Cutout/VertexLit"
---

和之前使用的Diffuse和Specular不同，这次我们使用内置的Transparent/Cutout/VertexLit来作为回调Shader。这不仅能够保证在我们编写的SubShader无法在当前显卡上工作时可以有合适的代替Shader，还可以保证使用透明度测试的物体可以正确地向其他物体投射阴影，具体原理可以参见9.4.5节。

材质面板中的Alpha cutoff参数用于调整透明度测试时使用的阈值，当纹理像素的透明度小于该值时，对应的片元就会被舍弃。当我们逐渐调大该值时，立方体上的网格会逐渐消失，如图8.7所示。





▲ 图8.7 随着Alpha cutoff参数的增大，更多的像素由于不满足透明度测试条件而被剔除

从图8.6和图8.7可以看出，透明度测试得到的透明效果很“极端”——要么完全透明，要么完全不透明，它的效果往往像在一个不透明物体上挖了一个空洞。而且，得到的透明效果在边缘处往往参差不齐，有锯齿，这是因为在边界处纹理的透明度的变化精度问题。为了得到更加柔滑的透明效果，就可以使用透明度混合。

## 8.4 透明度混合

透明度混合的实现要比透明度测试复杂一些，这是因为我们在处理透明度测试时，实际上跟对待普通的不透明物体几乎是一样的，只是在片元着色器中增加了对透明度判断并裁剪片元的代码。而要实现透明度混合就没有这么简单了。我们回顾之前提到的透明度混合的原理：

**透明度混合：**这种方法可以得到真正的半透明效果。它会使用当前片元的透明度作为混合因子，与已经存储在颜色缓冲中的颜色值进

行混合，得到新的颜色。但是，透明度混合需要关闭深度写入，这使得我们要非常小心物体的渲染顺序。

为了进行混合，我们需要使用Unity提供的混合命令——Blend。Blend是Unity提供的设置混合模式的命令。想要实现半透明的效果就需要把当前自身的颜色和已经存在于颜色缓冲中的颜色值进行混合，混合时使用的函数就是由该指令决定的。表8.2给出了Blend命令的语义。

表8.2 ShaderLab的Blend命令

语 义	描 述
Blend Off	关闭混合
Blend SrcFactor DstFactor	开启混合，并设置混合因子。源颜色（该片元产生的颜色）会乘以SrcFactor，而目标颜色（已经存在于颜色缓存的颜色）会乘以DstFactor，然后把两者相加后再存入颜色缓冲中
Blend SrcFactor DstFactor, SrcFactorA DstFactorA	和上面几乎一样，只是使用不同的因子来混合透明通道
BlendOp BlendOperation	并非是把源颜色和目标颜色简单相加后混合，而是使用BlendOperation对它们进行其他操作

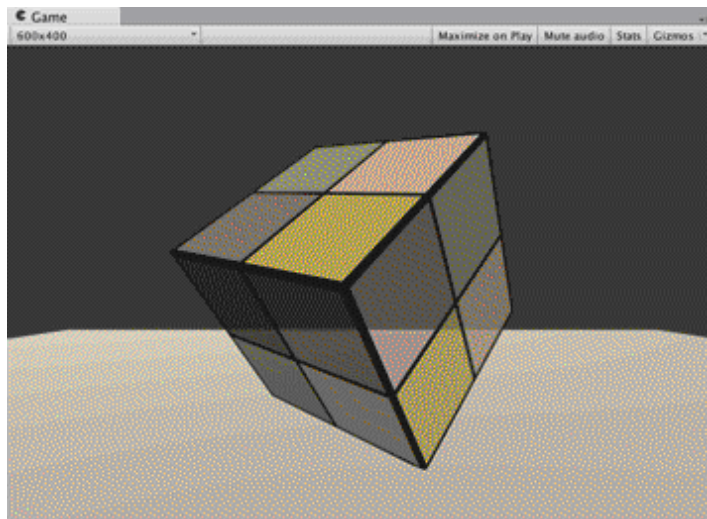
在本节里，我们会使用第二种语义，即Blend SrcFactor DstFactor来进行混合。需要注意的是，这个命令在设置混合因子的同时也开启了

混合模式。这是因为，只有开启了混合之后，设置片元的透明通道才有意义，而Unity在我们使用Blend命令的时候就自动帮我们打开了。很多初学者总是抱怨为什么自己的模型没有任何透明效果，这往往是因为他们没有在Pass中使用Blend命令，一方面是没有设置混合因子，但更重要的是，根本没有打开混合模式。我们会把源颜色的混合因子SrcFactor设为SrcAlpha，而目标颜色的混合因子DstFactor设为OneMinusSrcAlpha。这意味着，经过混合后新的颜色是：

$$DstColor_{new} = SrcAlpha \times SrcColor + (1 - SrcAlpha) \times DstColor_{old}$$

通常，透明度混合使用的就是这样的混合命令。在8.6节中，我们会看到更多混合语义的用法。

我们使用和8.3节中同样的透明纹理，在学习完本节后，我们可以得到类似图8.8这样的效果。



▲ 图8.8 透明度混合

为了在Unity中实现透明度混合，我们先进行如下准备工作。

(1) 在Unity中新建一个场景。在本书资源中，该场景名为Scene\_8\_4。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window -> Lighting -> Skybox中去掉场景中的天空盒子。

(2) 新建一个材质。在本书资源中，该材质名为AlphaBlendMat。

(3) 新建一个Unity Shader。在本书资源中，该Unity Shader名为Chapter8-AlphaBlend。把新的Unity Shader赋给第2步中创建的材质。

(4) 在场景中创建一个立方体，并把第2步中的材质赋给该模型。创建一个平面，使得平面位于立方体下面。

(5) 保存场景。

打开新建的Chapter8-AlphaBlend，删除所有已有代码，并把8.3节的Chapter8-AlphaTest代码全部粘贴进去，我们只需要在这个基础上进行一些修改即可。

(1) 修改Properties语义块：

```
Properties {  
    _Color ("Main Tint", Color) = (1,1,1,1)  
    _MainTex ("Main Tex", 2D) = "white" {}  
    _AlphaScale ("Alpha Scale", Range(0, 1)) = 1  
}
```

我们使用一个新的属性\_AlphaScale来替代原先的\_Cutoff属性。\_AlphaScale用于在透明纹理的基础上控制整体的透明度。相应的，我们也需要在Pass中修改和属性对应的变量：

```
fixed4 _Color;  
sampler2D _MainTex;  
float4 _MainTex_ST;  
fixed _AlphaScale;
```

(2) 修改SubShader使用的标签:

```
SubShader {  
    Tags { "Queue"="Transparent" "IgnoreProjector"="True"  
    "RenderType"="Transparent" }
```

在本章一开头，我们已经知道在Unity中透明度混合使用的渲染队列是名为Transparent的队列，因此我们需要把Queue标签设置为Transparent。RenderType标签可以让Unity把这个Shader归入到提前定义的组（这里就是Transparent组）中，用来指明该Shader是一个使用了透明度混合的Shader。RenderType标签通常被用于着色器替换功能。我们还把IgnoreProjector设置为True，这意味着这个Shader不会受到投影器（Projectors）的影响。通常，使用了透明度混合的Shader都应该在SubShader中设置这3个标签。

(3) 与透明度测试不同的是，我们还需要在Pass中为透明度混合进行合适的混合状态设置:

```
Pass {  
    Tags { "LightMode"="ForwardBase" }  
  
    ZWrite Off  
    Blend SrcAlpha OneMinusSrcAlpha
```

Pass的标签仍和之前一样，即把LightMode设为ForwardBase，这是为了让Unity能够按前向渲染路径的方式为我们正确提供各个光照变量。除此之外，我们还把该Pass的深度写入（ZWrite）设置为关闭状态（Off），我们在之前已经讲过为什么要这样做了。这是非常重要的。

然后，我们开启并设置了该Pass的混合模式。如在本节开头所讲的，我们将源颜色（该片元着色器产生的颜色）的混合因子设为SrcAlpha，把目标颜色（已经存在于颜色缓冲中的颜色）的混合因子设为OneMinusSrcAlpha，以得到合适的半透明效果。

#### (4) 修改片元着色器:

```
fixed4 frag(v2f i) : SV_Target {
    fixed3 worldNormal = normalize(i.worldNormal);
    fixed3 worldLightDir =
normalize(UnityWorldSpaceLightDir(i.worldPos));

    fixed4 texColor = tex2D(_MainTex, i.uv);

    fixed3 albedo = texColor.rgb * _Color.rgb;

    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;

    fixed3 diffuse = _LightColor0.rgb * albedo * max(0,
dot(worldNormal, worldLightDir));

    return fixed4(ambient + diffuse, texColor.a * _AlphaScale);
}
```

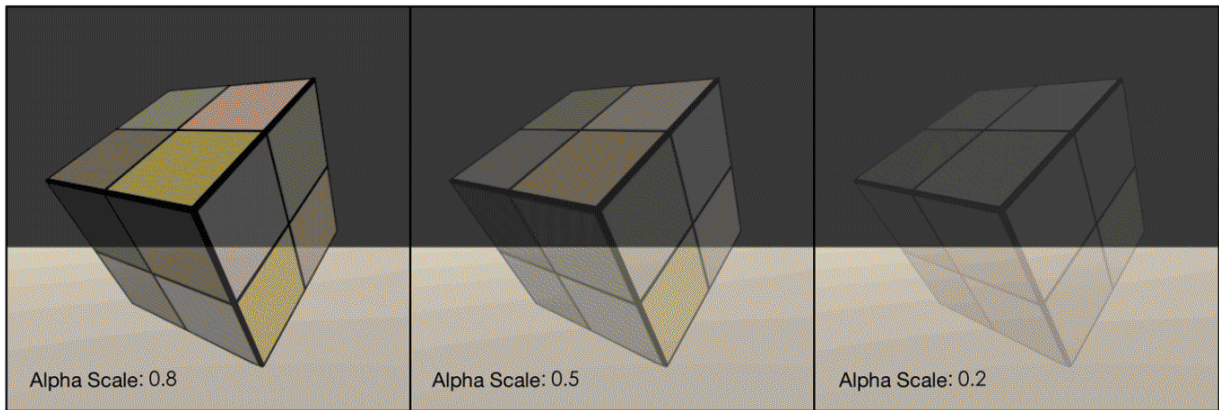
上述代码和8.3节中的几乎完全一样，只是移除了透明度测试的代码，并设置了该片元着色器返回值中的透明通道，它是纹理像素的透明通道和材质参数\_AlphaScale的乘积。正如本节一开始所说的，只有使用Blend命令打开混合后，我们在这里设置透明通道才有意义，否则，这些透明度并不会对片元的透明效果有任何影响。

#### (5) 最后，修改Unity Shader的Fallback:

```
Fallback "Transparent/VertexLit"
```

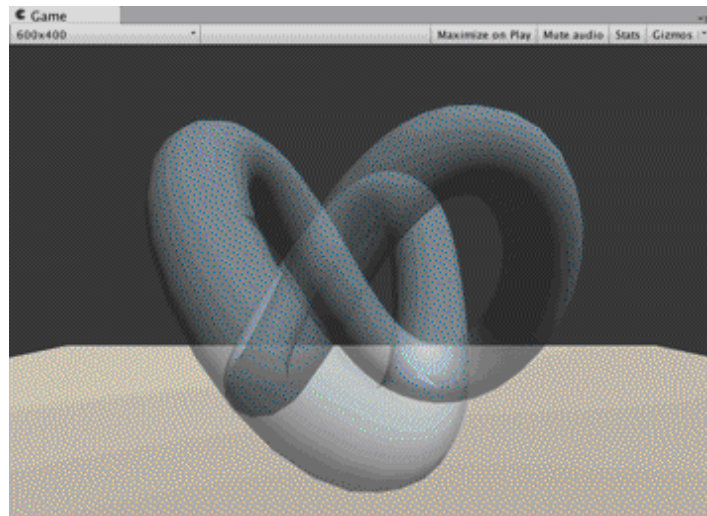
我们可以调节材质面板上的Alpha Scale参数，以控制整体透明度。图8.9给出了不同Alpha Scale参数下的半透明效果。





▲ 图8.9 随着Alpha Scale参数的减小，模型变得越来越透明

我们在8.1节中详细解释了由于关闭深度写入带来的各种问题。当模型本身有复杂的遮挡关系或是包含了复杂的非凸网格的时候，就会有各种各样因为排序错误而产生的错误的透明效果。图8.10给出了使用上面的Unity Shader渲染Knot模型时得到的效果。



▲ 图8.10 当模型网格之间有互相交叉的结构时，往往会得到错误的半透明效果

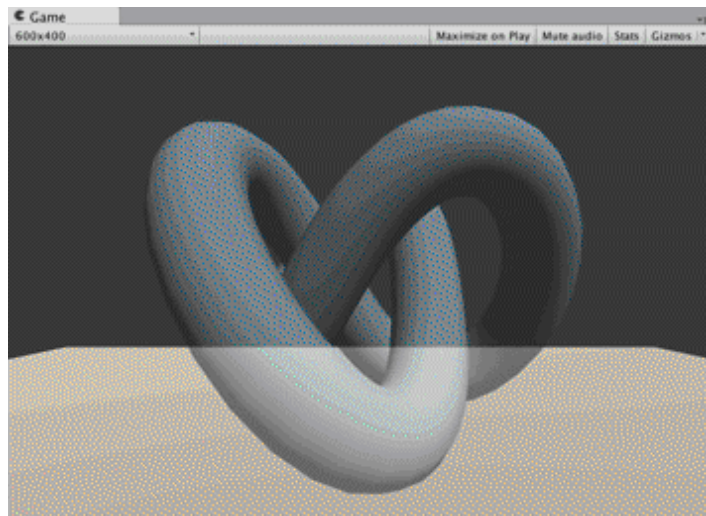
这都是由于我们关闭了深度写入造成的，因为这样我们就无法对模型进行像素级别的深度排序。在8.1节中我们提到了一种解决方法是



分割网格，从而可以得到一个“质量优等”的网格。但是很多情况下这往往是不切实际的。这时，我们可以想办法重新利用深度写入，让模型可以像半透明物体一样进行淡入淡出。这就是我们下面要讲的内容。

## 8.5 开启深度写入的半透明效果

在8.4节最后，我们给出了一种由于关闭深度写入而造成的错误排序的情况。一种解决方法是**使用两个Pass**来渲染模型：第一个Pass开启深度写入，但不输出颜色，它的目的仅仅是为了把该模型的深度值写入深度缓冲中；第二个Pass进行正常的透明度混合，由于上一个Pass已经得到了逐像素的正确的深度信息，该Pass就可以按照像素级别的深度排序结果进行透明渲染。但这种方法的缺点在于，多使用一个Pass会对性能造成一定的影响。在本节最后，我们可以得到类似图8.11中的效果。可以看出，使用这种方法，我们仍然可以实现模型与它后面的背景混合的效果，但模型内部之间不会有任何真正的半透明效果。



▲图8.11 开启了深度写入的半透明效果

为此，我们需要进行如下准备工作。

(1) 在Unity中新建一个场景。在本书资源中，该场景名为Scene\_8\_5。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window -> Lighting -> Skybox中去掉场景中的天空盒子。

(2) 新建一个材质。在本书资源中，该材质名为AlphaBlendZWriteMat。

(3) 新建一个Unity Shader。在本书资源中，该Unity Shader名为Chapter8-AlphaBlendZWrite。把新的Unity Shader赋给第2步中创建的材质。

(4) 在场景中创建一个立方体，并把第2步中的材质赋给该模型。创建一个平面，使得平面位于立方体下面。

(5) 保存场景。

本节使用的代码和8.4节使用的Chapter8-AlphaBlend几乎完全一样。我们把Chapter8-AlphaBlend中的代码粘贴到本节的Chapter8-AlphaBlendZWrite中，我们只需要在原来使用的Pass前面再增加一个新的Pass即可：

```
Shader "Unity Shader Book/Chapter8-Alpha Blending ZWrite" {
    Properties {
        _Color ("Main Tint", Color) = (1,1,1,1)
        _MainTex ("Main Tex", 2D) = "white" {}
        _AlphaScale ("Alpha Scale", Range(0, 1)) = 1
    }
}
```

```

SubShader {
    Tags {"Queue"="Transparent" "IgnoreProjector"="True"
"RenderType"="Transparent"}

    // Extra pass that renders to depth buffer only
    Pass {
        ZWrite On
        ColorMask 0
    }

    Pass {
        // 和8.4节同样的代码
    }
}
Fallback "Diffuse"
}

```

这个新添加的Pass的目的仅仅是为了把模型的深度信息写入深度缓冲中，从而剔除模型中被自身遮挡的片元。因此，Pass的第一行开启了深度写入。在第二行，我们使用了一个新的渲染命令——**ColorMask**。在ShaderLab中，ColorMask用于设置颜色通道的写掩码（write mask）。它的语义如下：

ColorMask	RGB		A		0		其他任何R、G、B、A的组合
-----------	-----	--	---	--	---	--	----------------

当ColorMask设为0时，意味着该Pass不写入任何颜色通道，即不会输出任何颜色。这正是我们需要的——该Pass只需写入深度缓存即可。

## 8.6 ShaderLab的混合命令

在8.4一节中，我们已经看到如何利用Blend命令进行混合。实际上，混合还有很多其他用处，不仅仅是用于透明度混合。在本节里，我们将更加详细地了解混合中的细节问题。

我们首先来看一下混合是如何实现的。当片元着色器产生一个颜色的时候，可以选择与颜色缓存中的颜色进行混合。这样一来，混合就和两个操作数有关：**源颜色（source color）**和**目标颜色**

**（destination color）**。源颜色，我们用**S**表示，指的是由片元着色器产生的颜色值；目标颜色，我们用**D**表示，指的是从颜色缓冲中读取到的颜色值。对它们进行混合后得到的输出颜色，我们用**O**表示，它会重新写入到颜色缓冲中。需要注意的是，当我们谈及混合中的源颜色、目标颜色和输出颜色时，它们都包含了**RGBA**四个通道的值，而并非仅仅是**RGB**通道。

想要使用混合，我们必须首先开启它。在Unity中，当我们使用Blend（Blend Off命令除外）命令时，除了设置混合状态外也开启了混合。但是，在其他图形API中我们是需要手动开启的。例如在OpenGL中，我们需要使用glEnable(GL\_BLEND)来开启混合。但在Unity中，它已经在背后为我们做了这些工作。

### 8.6.1 混合等式和参数

在2.3.8节中我们提到过，混合是一个逐片元的操作，而且它不是可编程的，但却是高度可配置的。也就是说，我们可以设置混合时使用的运算操作、混合因子等来影响混合。那么，这些配置又是如何实现的呢？

现在，我们已知两个操作数：源颜色**S**和目标颜色**D**，想要得到输出颜色**O**就必须使用一个等式来计算。我们把这个等式称为**混合等式（blend equation）**。当进行混合时，我们需要使用两个混合等式：一个用于混合**RGB**通道，一个用于混合**A**通道。当设置混合状态时，我们

实际上设置的就是混合等式中的**操作**和**因子**。在默认情况下，混合等式使用的操作都是加操作（我们也可以使用其他操作），我们只需要再设置一下混合因子即可。由于需要两个等式（分别用于混合**RGB**通道和**A**通道），每个等式有两个因子（一个用于和源颜色相乘，一个用于和目标颜色相乘），因此一共需要4个因子。表8.3给出了ShaderLab中设置混合因子的命令。

表8.3 ShaderLab中设置混合因子的命令

命 令	描 述
Blend SrcFactor DstFactor	开启混合，并设置混合因子。源颜色（该片元产生的颜色）会乘以SrcFactor，而目标颜色（已经存在于颜色缓存的颜色）会乘以DstFactor，然后把两者相加后再存入颜色缓冲中
Blend SrcFactor DstFactor, SrcFactorA DstFactorA	和上面几乎一样，只是使用不同的因子来混合透明通道

可以发现，第一个命令只提供了两个因子，这意味着将使用同样的混合因子来混合**RGB**通道和**A**通道，即此时SrcFactorA将等于SrcFactor，DstFactorA将等于DstFactor。下面就是使用这些因子进行加法混合时使用的混合公式：

$$O_{rgb} = SrcFactor \times S_{rgb} + DstFactor \times D_{rgb}$$

$$O_a = SrcFactorA \times S_a + DstFactorA \times D_a$$

那么，这些混合因子可以有哪些值呢？表8.4给出了ShaderLab支持的几种混合因子。

表8.4 ShaderLab中的混合因子

参 数	描 述
One	因子为1
Zero	因子为0
SrcColor	因子为源颜色值。当用于混合RGB的混合等式时，使用SrcColor的RGB分量作为混合因子；当用于混合A的混合等式时，使用SrcColor的A分量作为混合因子
SrcAlpha	因子为源颜色的透明度值（A通道）
DstColor	因子为目标颜色值。当用于混合RGB通道的混合等式时，使用DstColor的RGB分量作为混合因子；当用于混合A通道的混合等式时，使用DstColor的A分量作为混合因子。
DstAlpha	因子为目标颜色的透明度值（A通道）
OneMinusSrcColor	因子为(1-源颜色)。当用于混合RGB的混合等式时，使用结果的RGB分量作为混合因子；当用于混合A的混合等式时，使用结果的A分量作为混合因子
OneMinusSrcAlpha	因子为(1-源颜色的透明度值)

参 数	描 述
OneMinusDstColor	因子为(1-目标颜色)。当用于混合RGB的混合等式时，使用结果的RGB分量作为混合因子；当用于混合A的混合等式时，使用结果的A分量作为混合因子
OneMinusDstAlpha	因子为(1-目标颜色的透明度值)

使用上面的指令进行设置时，RGB通道的混合因子和A通道的混合因子都是一样的，有时我们希望可以使用不同的参数混合A通道，这时就可以利用**Blend SrcFactor DstFactor, SrcFactorA DstFactorA**指令。例如，如果我们想要在混合后，输出颜色的透明度值就是源颜色的透明度，可以使用下面的命令：

```
Blend SrcAlpha OneMinusSrcAlpha, One Zero
```

8.6.2 混合操作

在上面涉及的混合等式中，当把源颜色和目标颜色与它们对应的混合因子相乘后，我们都是把它们的结果加起来作为输出颜色的。那么可不可以选择不使用加法，而使用减法呢？答案是肯定的，我们可以使用ShaderLab的**BlendOp BlendOperation**命令，即混合操作命令。表8.5给出了ShaderLab中支持的混合操作。

表8.5 ShaderLab中的混合操作



操作	描述
Add	将混合后的源颜色和目标颜色相加。默认的混合操作。使用的混合等式是： $O_{rgb} = SrcFactor \times S_{rgb} + DstFactor \times D_{rgb}$ $O_a = SrcFactor A \times S_a + DstFactor A \times D_a$
Sub	用混合后的源颜色减去混合后的目标颜色。使用的混合等式是： $O_{rgb} = SrcFactor \times S_{rgb} - DstFactor \times D_{rgb}$ $O_a = SrcFactor A \times S_a - DstFactor A \times D_a$
RevSub	用混合后的目标颜色减去混合后的源颜色。使用的混合等式是： $O_{rgb} = DstFactor \times D_{rgb} - SrcFactor \times S_{rgb}$ $O_a = DstFactor A \times D_a - SrcFactor A \times S_a$
Min	使用源颜色和目标颜色中较小的值，是逐分量比较的。使用的混合等式是： $O_{rgba} = (\min(S_r, D_r), \min(S_g, D_g), \min(S_b, D_b), \min(S_a, D_a))$
Max	使用源颜色和目标颜色中较大的值，是逐分量比较的。使用的混合等式是： $O_{rgba} = (\max(S_r, D_r), \max(S_g, D_g), \max(S_b, D_b), \max(S_a, D_a))$
其他逻辑操作	仅在DirectX 11.1中支持

混合操作命令通常是与混合因子命令一起工作的。但需要注意的是，当使用**Min**或**Max**混合操作时，混合因子实际上是不起任何作用的，它们仅会判断原始的源颜色和目的颜色之间的比较结果。

### 8.6.3 常见的混合类型

通过混合操作和混合因子命令的组合，我们可以得到一些类似Photoshop混合模式中的混合效果：

```
// 正常 (Normal)，即透明度混合
Blend SrcAlpha OneMinusSrcAlpha

// 柔和相加 (Soft Additive)
Blend OneMinusDstColor One

// 正片叠底 (Multiply)，即相乘
Blend DstColor Zero

// 两倍相乘 (2x Multiply)
Blend DstColor SrcColor

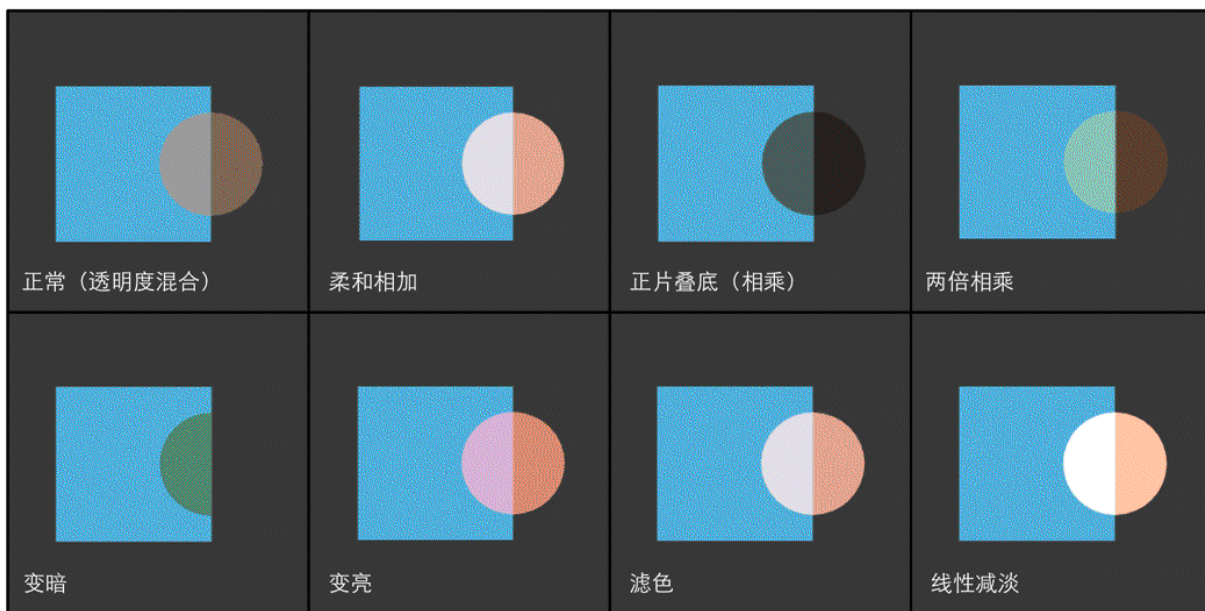
// 变暗 (Darken)
BlendOp Min
Blend One One

// 变亮 (Lighten)
BlendOp Max
Blend One One

// 滤色 (Screen)
Blend OneMinusDstColor One
// 等同于
Blend One OneMinusSrcColor

// 线性减淡 (Linear Dodge)
Blend One One
```

图8.12给出了上面不同设置下得到的结果。我们可以在本书资源中的Scene\_8\_6\_3场景中找到相关资源。



▲ 图8.12 不同混合状态设置得到的效果

需要注意的是，虽然上面使用**Min**和**Max**混合操作时仍然设置了混合因子，但实际上它们并不会对结果有任何影响，因为**Min**和**Max**混合操作会忽略混合因子。另一点是，虽然上面有些混合模式并没有设置混合操作的种类，但是它们默认就是使用加法操作，相当于设置了BlendOp Add。

## 8.7 双面渲染的透明效果

在现实生活中，如果一个物体是透明的，意味着我们不仅可以透过它看到其他物体的样子，也可以看到它内部的结构。但在前面实现的透明效果中，无论是透明度测试还是透明度混合，我们都无法观察到正方体内部及其背面的形状，导致物体看起来就好像只有半个一样。这是因为，默认情况下渲染引擎剔除了物体背面（相对于摄像机的方向）的渲染图元，而只渲染了物体的正面。如果我们想要得到双

面渲染的效果，可以使用**Cull**指令来控制需要剔除哪个面的渲染图元。在Unity中，**Cull**指令的语法如下：

<code>Cull Back   Front   Off</code>
--------------------------------------

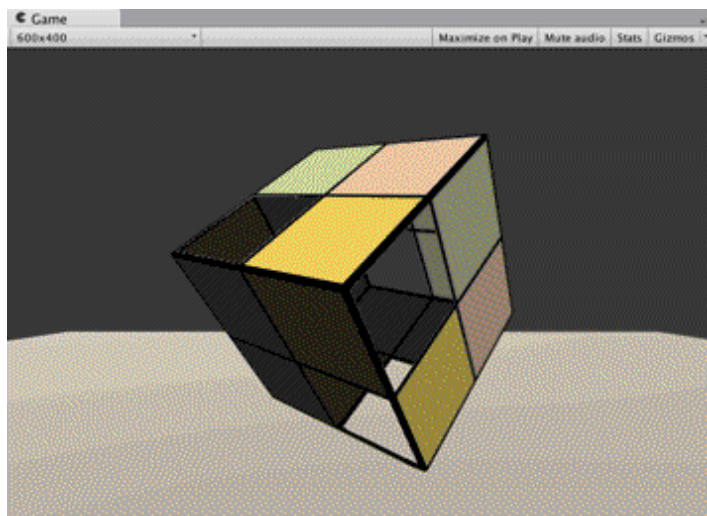
如果设置为**Back**，那么那些背对着摄像机的渲染图元就不会被渲染，这也是默认情况下的剔除状态；如果设置为**Front**，那么那些朝向摄像机的渲染图元就不会被渲染；如果设置为**Off**，就会关闭剔除功能，那么所有的渲染图元都会被渲染，但由于这时需要渲染的图元数目会成倍增加，因此除非是用于特殊效果，例如这里的双面渲染的透明效果，通常情况是不会关闭剔除功能的。

### 8.7.1 透明度测试的双面渲染

我们首先来看一下，如何让使用了透明度测试的物体实现双面渲染的效果。这非常简单，只需要在**Pass**的渲染设置中使用**Cull**指令来关闭剔除即可。为此，我们新建了一个场景，在本章资源中，该场景名为**Scene\_8\_7\_1**，场景中同样包含了一个正方体，它使用的材质和Unity Shader分别名为**AlphaTestBothSidedMat**和**Chapter8-AlphaTestBothSided**。Chapter8-AlphaTestBothSided的代码和8.3节中的Chapter8-AlphaTest几乎完全一样，只添加了一行代码：

<pre>Pass {     Tags { "LightMode"="ForwardBase" }      // Turn off culling     Cull Off</pre>
--

如上所示，这行代码的作用是关闭剔除功能，使得该物体的所有的渲染图元都会被渲染。由此，我们可以得到图8.13中的效果。



▲图8.13 双面渲染的透明度测试的物体

此时，我们可以透过正方体的镂空区域看到内部的渲染结果。

## 8.7.2 透明度混合的双面渲染

和透明度测试相比，想要让透明度混合实现双面渲染会更复杂一些，这是因为透明度混合需要关闭深度写入，而这是“一切混乱的开端”。我们知道，想要得到正确的透明效果，渲染顺序是非常重要的——我们想要保证图元是从后往前渲染的。对于透明度测试来说，由于我们没有关闭深度写入，因此可以利用深度缓冲按逐像素的粒度进行深度排序，从而保证渲染的正确性。然而一旦关闭了深度写入，我们就需要小心地控制渲染顺序来得到正确的深度关系。如果我们仍然采用8.7.1节中的方法，直接关闭剔除功能，那么我们就无法保证同一个物体的正面和背面图元的渲染顺序，就有可能得到错误的半透明效果。

为此，我们选择把双面渲染的工作分成两个Pass——第一个Pass只渲染背面，第二个Pass只渲染正面，由于Unity会顺序执行SubShader中

的各个Pass，因此我们可以保证背面总是在正面被渲染之前渲染，从而可以保证正确的深度渲染关系。

我们新建了一个场景，在本章资源中，该场景名为Scene\_8\_7\_2，场景中包含了一个正方体，它使用的材质和Unity Shader分别名为AlphaBlendBothSidedMat和Chapter8-AlphaBlendBothSided。相较于8.4节的Chapter8-AlphaBlend，我们对Chapter8-AlphaTestBothSided的代码做了两个改动。

(1) 复制原Pass的代码，得到另一个Pass。

(2) 在两个Pass中分别使用Cull指令剔除不同朝向的渲染图元：

```
Shader "Unity Shaders Book/Chapter 8/Alpha Blend With Both Side" {
    Properties {
        _Color ("Main Tint", Color) = (1,1,1,1)
        _MainTex ("Main Tex", 2D) = "white" {}
        _AlphaScale ("Alpha Scale", Range(0, 1)) = 1
    }
    SubShader {
        Tags {"Queue"="Transparent" "IgnoreProjector"="True"
"RenderType"="Transparent"}

        Pass {
            Tags { "LightMode"="ForwardBase" }

            // First pass renders only back faces
            Cull Front

            // 和之前一样的代码
        }

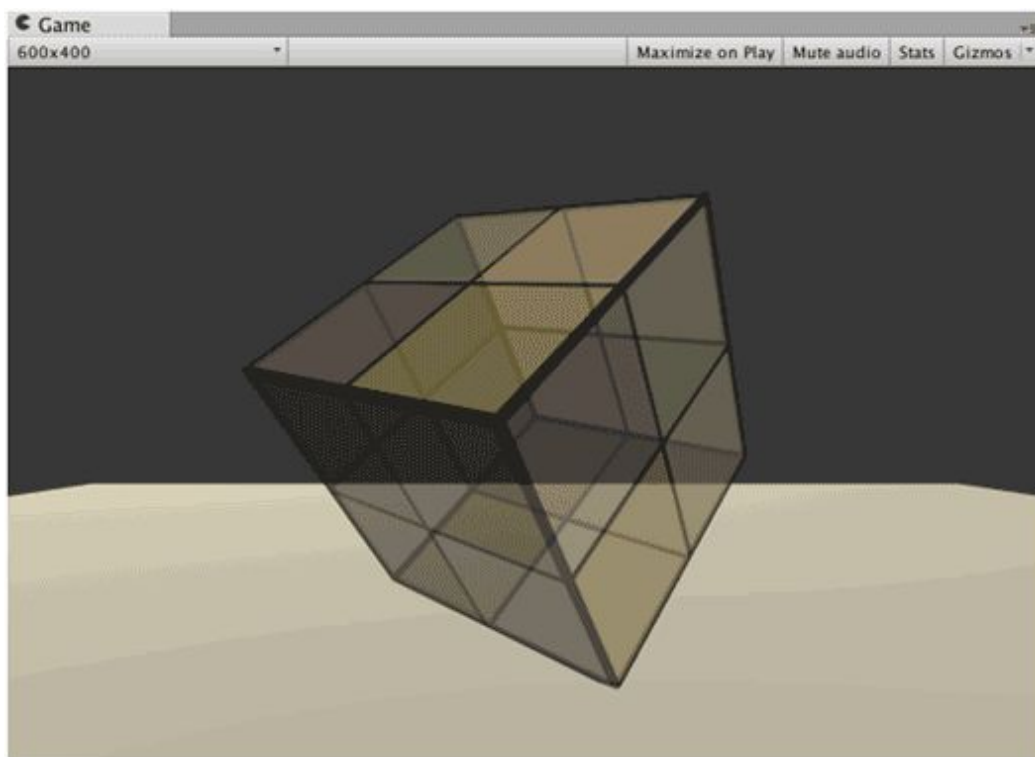
        Pass {
            Tags { "LightMode"="ForwardBase" }

            // Second pass renders only front faces
            Cull Back

            // 和之前一样的代码
        }
    }
}
```

```
} Fallback "Transparent/VertexLit"
```

通过上面的代码，我们可以得到图8.14中的效果。



▲图8.14 双面渲染的透明度混合的物体



## 第3篇 中级篇

中级篇是本书的进阶篇，将讲解Unity中的渲染路径、如何计算光照衰减和阴影、如何使用高级纹理和动画等一系列进阶内容。

### 第9章 更复杂的光照

我们在初级篇中实现的光照模型中没有考虑一些重要的光照计算，如阴影和光照衰减。本章首先讲解Unity中的3种渲染路径和3种重要的光源类型，再解释如何在前向渲染路径中实现包含了光照衰减、阴影等效果的完整的光照计算。在本章最后，会给出基于之前学习内容实现的包含了完整光照计算的Unity Shader。

### 第10章 高级纹理

这一章将会讲解如何在Unity Shader中使用立方体纹理、渲染纹理和程序纹理等类型的纹理。

### 第11章 让画面动起来

静态的画面往往是无趣的。这一章将帮助读者学习如何在Shader中使用时间变量来实现纹理动画、顶点动画等动态效果。

## 第9章 更复杂的光照

从本章开始，我们就进入了中级篇的学习。在初级篇中，我们对实现的Unity Shader中的每一行代码都进行了详细解释。我们相信通过初级篇的学习，读者已经对Shader的基本语法有了一定了解，因此在中级篇以及之后的篇节中，我们不再列出Unity Shader中的每一行代码，而是选择其中的关键代码进行解释。读者可以在本书资源中找到完整的实现。**需要注意的是**，本章实现的代码大多是为了阐述一些计算的实现原理，并不可以直接用于项目中。我们会在9.5节给出包含了完整光照计算的Unity Shader。

在前面的学习中，我们的场景中都仅有一个光源且光源类型是平行光（如果你的场景不是这样的话，可能会得到错误的结果）。但在实际的游戏开发过程中，我们往往需要处理数目更多、类型更复杂的光源。更重要的是，我们想要得到阴影。在本章我们就会学习如何在Unity中实现上面的功能。

在学习这些之前，我们有必要知道Unity到底是如何处理这些光源的。也就是说，当我们在场景里放置了各种类型的光源后，Unity的底层渲染引擎是如何让我们在Shader中访问到它们的，因此9.1节首先介绍了Unity的渲染路径。之后，我们将在9.2节中学习如何处理更多不同类型的光源，如点光源和聚光灯。9.3节将介绍如何在Unity Shader中处理光照衰减，实现距离光源越远光强越弱的效果。在9.4节，我们将介绍Unity中阴影的实现方法，并学习在Unity Shader中如何为不同类型的

物体实现阴影效果。最后，我们会在9.5节给出本书使用的标准的Unity Shader，这些Unity Shader包含了完整的光照计算，本书后面的章节中也会使用这些Shader进行场景搭建。

## 9.1 Unity的渲染路径

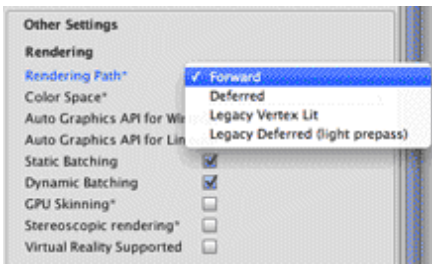
在Unity里，**渲染路径（Rendering Path）**决定了光照是如何应用到Unity Shader中的。因此，如果要和光源打交道，我们需要为每个Pass指定它使用的渲染路径，只有这样才能让Unity知道，“哦，原来这个程序员想要这种渲染路径，那么好的，我把光源和处理后的光照信息都放在这些数据里，你可以访问啦！”也就是说，我们只有为Shader正确地选择和设置了需要的渲染路径，该Shader的光照计算才能被正确执行。

Unity支持多种类型的渲染路径。在Unity 5.0版本之前，主要有3种：**前向渲染路径（Forward Rendering Path）**、**延迟渲染路径（Deferred Rendering Path）**和**顶点照明渲染路径（Vertex Lit Rendering Path）**。但在Unity 5.0版本以后，Unity做了很多更改，主要有两个变化：首先，顶点照明渲染路径已经被Unity抛弃（但目前仍然可以对之前使用了顶点照明渲染路径的Unity Shader兼容）；其次，新的延迟渲染路径代替了原来的延迟渲染路径（同样，目前也提供了对较旧版本的兼容）。

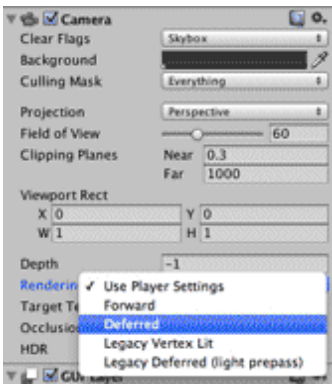
大多数情况下，一个项目只使用一种渲染路径，因此我们可以为整个项目设置渲染时的渲染路径。我们可以通过在Unity的Edit → Project Settings → Player → Other Settings → Rendering Path中选择项目

所需的渲染路径。默认情况下，该设置选择的是前向渲染路径，如图9.1所示。

但有时，我们希望可以同时使用多个渲染路径，例如摄像机A渲染的物体使用前向渲染路径，而摄像机B渲染的物体使用延迟渲染路径。这时，我们可以在每个摄像机的渲染路径设置中设置该摄像机使用的渲染路径，以覆盖Project Settings中的设置，如图9.2所示。



▲ 图9.1 设置Unity项目的渲染路径



▲ 图9.2 摄像机组件的Rendering Path中的设置可以覆盖Project Settings中的设置

在上面的设置中，如果选择了Use Player Settings，那么这个摄像机会使用Project Settings中的设置；否则就会覆盖掉Project Settings中的设置。需要注意的是，如果当前的显卡并不支持所选择的渲染路径，

Unity会自动使用更低一级的渲染路径。例如，如果一个GPU不支持延迟渲染，那么Unity就会使用前向渲染。

完成了上面的设置后，我们就可以在每个Pass中使用标签来指定该Pass使用的渲染路径。这是通过设置Pass的**LightMode**标签实现的。不同类型的渲染路径可能会包含多种标签设置。例如，我们之前在代码中写的：

```
Pass {  
    Tags { "LightMode" = "ForwardBase" }  
}
```

上面的代码将告诉Unity，该Pass使用前向渲染路径中的**ForwardBase**路径。而前向渲染路径还有一种路径叫做**ForwardAdd**。表9.1给出了Pass的LightMode标签支持的渲染路径设置选项。

表9.1      **LightMode**标签支持的渲染路径设置选项

标 签 名	描 述
Always	不管使用哪种渲染路径，该Pass总是会被渲染，但不会计算任何光照
ForwardBase	用于 <b>前向渲染</b> 。该Pass会计算环境光、最重要的平行光、逐顶点/SH光源和Lightmaps
ForwardAdd	用于 <b>前向渲染</b> 。该Pass会计算额外的逐像素光源，每个Pass对应一个光源
Deferred	用于 <b>延迟渲染</b> 。该Pass会渲染G缓冲（G-buffer）

标 签 名	描 述
ShadowCaster	把物体的深度信息渲染到阴影映射纹理（shadowmap）或一张深度纹理中
PrepassBase	用于 <b>遗留的延迟渲染</b> 。该Pass会渲染法线和高光反射的指数部分
PrepassFinal	用于 <b>遗留的延迟渲染</b> 。该Pass通过合并纹理、光照和自发光来渲染得到最后颜色
Vertex 、 VertexLMRGBM和 VertexLM	用于 <b>遗留的顶点照明渲染</b>

那么指定渲染路径到底有什么用呢？如果一个Pass没有指定任何渲染路径会有什么问题吗？通俗来讲，指定渲染路径是我们和Unity的底层渲染引擎的一次重要的沟通。例如，如果我们为一个Pass设置了前向渲染路径的标签，相当于会告诉Unity：“嘿，我准备使用前向渲染了，你把那些光照属性都按前向渲染的流程给我准备好，我一会儿要用！”随后，我们可以通过Unity提供的内置光照变量来访问这些属性。如果我们没有指定任何渲染路径（实际上，在Unity 5.x版本中如果使用了前向渲染又没有为Pass指定任何前向渲染适合的标签，就会被当成一

个和顶点照明渲染路径等同的Pass)，那么一些光照变量很可能不会被正确赋值，我们计算出的效果也就很有可能是错误的。

那么，Unity的渲染引擎是如何处理这些渲染路径的呢？下面，我们会对这些渲染路径进行更加详细的解释。

### 9.1.1 前向渲染路径

前向渲染路径是传统的渲染方式，也是我们最常用的一种渲染路径。在本节，我们首先会概括前向渲染路径的原理，然后再给出Unity对于前向渲染路径的实现细节和要求，最后给出Unity Shader中哪些内置变量是用于前向渲染路径的。

#### 1. 前向渲染路径的原理

每进行一次完整的前向渲染，我们需要渲染该对象的渲染图元，并计算两个缓冲区的信息：一个是颜色缓冲区，一个是深度缓冲区。我们利用深度缓冲来决定一个片元是否可见，如果可见就更新颜色缓冲区中的颜色值。我们可以用下面的伪代码来描述前向渲染路径的大致过程：

```
Pass {
    for (each primitive in this model) {
        for (each fragment covered by this primitive) {
            if (failed in depth test) {
                // 如果没有通过深度测试，说明该片元是不可见的
                discard;
            } else {
                // 如果该片元可见
                // 就进行光照计算
                float4 color = Shading(materialInfo, pos, normal,
lightDir, viewDir);
                // 更新帧缓冲
                writeFrameBuffer(fragment, color);
            }
        }
    }
}
```



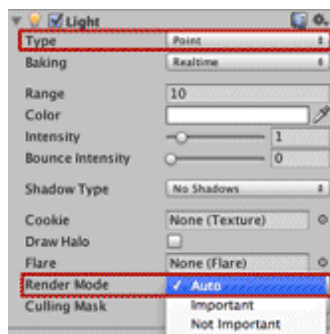
```
}  
}  
}
```

对于每个逐像素光源，我们都需要进行上面一次完整的渲染流程。如果一个物体在多个逐像素光源的影响区域内，那么该物体就需要执行多个Pass，每个Pass计算一个逐像素光源的光照结果，然后在帧缓冲中把这些光照结果混合起来得到最终的颜色值。假设，场景中有 $N$ 个物体，每个物体受 $M$ 个光源的影响，那么要渲染整个场景一共需要 $N*M$ 个Pass。可以看出，如果有大量逐像素光照，那么需要执行的Pass数目也会很大。因此，渲染引擎通常会限制每个物体的逐像素光照的数目。

## 2. Unity中的前向渲染

事实上，一个Pass不仅仅可以用来计算逐像素光照，它也可以用来计算逐顶点等其他光照。这取决于光照计算所处流水线阶段以及计算时使用的数学模型。当我们渲染一个物体时，Unity会计算哪些光源照亮了它，以及这些光源照亮该物体的方式。

在Unity中，前向渲染路径有3种处理光照（即照亮物体）的方式：**逐顶点处理、逐像素处理，球谐函数（Spherical Harmonics, SH）处理**。而决定一个光源使用哪种处理模式取决于它的类型和渲染模式。光源类型指的是该光源是平行光还是其他类型的光源，而光源的渲染模式指的是该光源是否是**重要的（Important）**。如果我们把一个光照的模式设置为Important，意味着我们告诉Unity，“嘿老兄，这个光源很重要，我希望你可以认真对待它，把它当成一个逐像素光源来处理！”我们可以在光源的Light组件中设置这些属性，如图9.3所示。

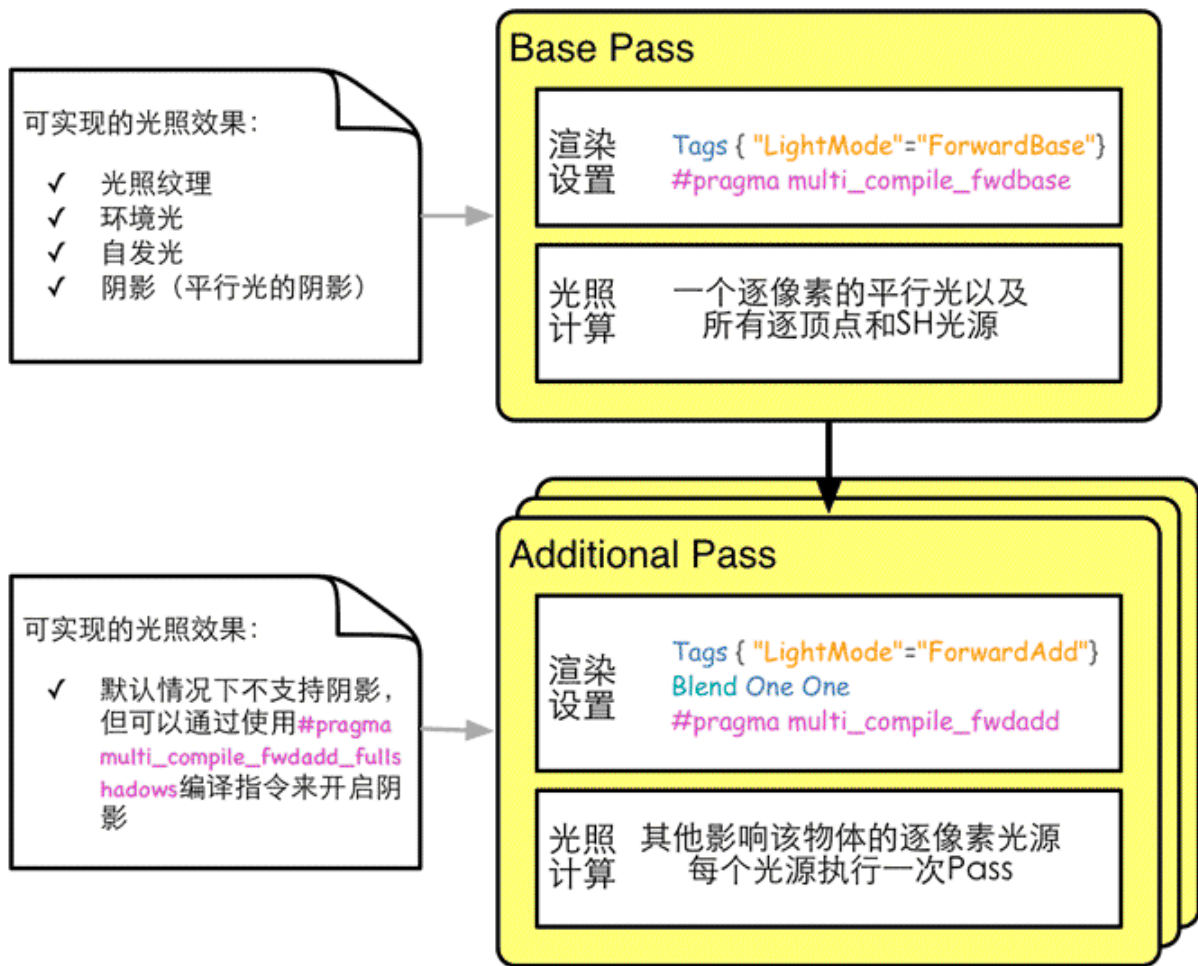


▲ 图9.3 设置光源的类型和渲染模式

在前向渲染中，当我们渲染一个物体时，Unity会根据场景中各个光源的设置以及这些光源对物体的影响程度（例如，距离该物体的远近、光源强度等）对这些光源进行一个重要度排序。其中，一定数目的光源会按逐像素的方式处理，然后最多有4个光源按逐顶点的方式处理，剩下的光源可以按SH方式处理。Unity使用的判断规则如下。

- 场景中最亮的平行光总是按逐像素处理的。
- 渲染模式被设置成**Not Important**的光源，会按逐顶点或者SH处理。
- 渲染模式被设置成**Important**的光源，会按逐像素处理。
- 如果根据以上规则得到的逐像素光源数量小于**Quality Setting**中的逐像素光源数量(**Pixel Light Count**)，会有更多的光源以逐像素的方式进行渲染。

那么，在哪里进行光照计算呢？当然是在Pass里。前面提到过，前向渲染有两种Pass：**Base Pass**和**Additional Pass**。通常来说，这两种Pass进行的标签和渲染设置以及常规光照计算如图9.4所示。



▲ 图9.4 前向渲染的两种Pass

图9.4中有几点需要说明的地方。

- 首先，可以发现在渲染设置中，我们除了设置了Pass的标签外，还使用了 `#pragma multi_compile_fwdbase` 这样的编译指令。根据官方文档（<https://docs.unity3d.com/Manual/SL-MultipleProgramVariants.html>）中的相关解释，我们可以知道，这些编译指令会保证Unity可以为相应类型的Pass生成所有需要的Shader变种，这些变种会处理不同条件下的渲染逻辑，例如是否使

用光照贴图、当前处理哪种光源类型、是否开启了阴影等，同时Unity也会在背后声明相关的内置变量并传递到Shader中。通俗来讲，只有分别为Base Pass和Additional Pass使用这两个编译指令，我们才可以在相关的Pass中得到一些正确的光照变量，例如光照衰减等。

- Base Pass旁边的注释给出了Base Pass中支持的一些光照特性。例如在Base Pass中，我们可以访问光照纹理（lightmap）。
- Base Pass中渲染的平行光默认是支持阴影的（如果开启了光源的阴影功能），而Additional Pass中渲染的光源在默认情况下是没有阴影效果的，即便我们在它的Light组件中设置了有阴影的Shadow Type。但我们可以在Additional Pass中使用 `#pragma multicompile fwdadd_fullshadows` 代替 `#pragma multi_compile_fwdadd` 编译指令，为点光源和聚光灯开启阴影效果，但这需要Unity在内部使用更多的Shader变种。
- 环境光和自发光也是在Base Pass中计算的。这是因为，对于一个物体来说，环境光和自发光我们只希望计算一次即可，而如果我们在Additional Pass中计算这两种光照，就会造成叠加多次环境光和自发光，这不是我们想要的。
- 在Additional Pass的渲染设置中，我们还开启和设置了混合模式。这是因为，我们希望每个Additional Pass可以与上一次的光照结果在帧缓存中进行叠加，从而得到最终的有多个光照的渲染效果。如果我们没有开启和设置混合模式，那么Additional Pass的渲染结果会覆盖掉之前的渲染结果，看起来就好像该物体只受该光源的影响。通常情况下，我们选择的混合模式是Blend One One。

- 对于前向渲染来说，一个Unity Shader通常会定义一个Base Pass（Base Pass也可以定义多次，例如需要双面渲染等情况）以及一个Additional Pass。一个Base Pass仅会执行一次（定义了多个Base Pass的情况除外），而一个Additional Pass会根据影响该物体的其他逐像素光源的数目被多次调用，即每个逐像素光源会执行一次Additional Pass。

图9.4给出的光照计算是**通常情况**下我们在每种Pass中进行的计算。实际上，渲染路径的设置用于告诉Unity该Pass在前向渲染路径中的位置，然后底层的渲染引擎会进行相关计算并填充一些内置变量（如\_LightColor0等），如何使用这些内置变量进行计算完全取决于开发者的选择。例如，我们完全可以利用Unity提供的内置变量在Base Pass中只进行逐顶点光照；同样，我们也完全可以在Additional Pass中按逐顶点的方式进行光照计算，不进行任何逐像素光照计算。

### 3. 内置的光照变量和函数

前面说过，根据我们使用的渲染路径（即Pass标签中LightMode的值），Unity会把不同的光照变量传递给Shader。

在Unity 5中，对于前向渲染（即LightMode为ForwardBase或ForwardAdd）来说，表9.2给出了我们可以在Shader中访问到的光照变量。

表9.2 前向渲染可以使用的内置光照变量

名 称	类 型	描 述

名 称	类 型	描 述
<code>_LightColor0</code>	<code>float4</code>	该Pass处理的逐像素光源的颜色
<code>_WorldSpaceLightPos0</code>	<code>float4</code>	<code>_WorldSpaceLightPos0.xyz</code> 是该Pass处理的逐像素光源的位置。如果该光源是平行光，那么 <code>_WorldSpaceLightPos0.w</code> 是0，其他光源类型w值为1
<code>_LightMatrix0</code>	<code>float4×4</code>	从世界空间到光源空间的变换矩阵。可以用于采样cookie和光强衰减（attenuation）纹理
<code>unity_4LightPosX0</code> , <code>unity_4LightPosY0</code> , <code>unity_4LightPosZ0</code>	<code>float4</code>	仅用于Base Pass。前4个非重要的点光源在世界空间中的位置
<code>unity_4LightAtten0</code>	<code>float4</code>	仅用于Base Pass。存储了前4个非重要的点光源的衰减因子
<code>unity_LightColor</code>	<code>half4[4]</code>	仅用于Base Pass。存储了前4个非重要的点光源的颜色

我们在6.6节中已经给出了一些可以用于前向渲染路径的函数，例如WorldSpaceLightDir、UnityWorldSpaceLightDir和ObjSpaceLightDir。

为了完整性，我们在表9.3中再次列出了前向渲染中可以使用的内置光照函数。

表9.3 前向渲染可以使用的内置光照函数

函 数 名	描 述
float3 WorldSpaceLightDir (float4 v)	仅可用于前向渲染中。输入一个模型空间中的顶点位置，返回世界空间中从该点到光源的光照方向。内部实现使用了UnityWorldSpaceLightDir函数。没有被归一化
float3 UnityWorldSpaceLightDir (float4 v)	仅可用于前向渲染中。输入一个世界空间中的顶点位置，返回世界空间中从该点到光源的光照方向。没有被归一化
float3 ObjSpaceLightDir (float4 v)	仅可用于前向渲染中。输入一个模型空间中的顶点位置，返回模型空间中从该点到光源的光照方向。没有被归一化
float3 Shade4PointLights (...)	仅可用于前向渲染中。计算四个点光源的光照，它的参数是已经打包进矢量的光照数据，通常就是表9.2中的内置变量，如unity_4LightPosX0, unity_4LightPosY0, unity_4LightPosZ0、unity_LightColor和unity_4LightAtten0等。前向渲染通常会使用这个函数来计算逐顶点光照

需要说明的是，上面给出的变量和函数并不是完整的，一些前向渲染可以使用的内置变量和函数官方文档中并没有给出说明。在后面的学习中，我们会使用到一些不在这些表中的变量和函数，那时我们会特别说明的。



## 9.1.2 顶点照明渲染路径

顶点照明渲染路径是对硬件配置要求最少、运算性能最高，但同时也是得到的效果最差的一种类型，它不支持那些逐像素才能得到的效果，例如阴影、法线映射、高精度的高光反射等。实际上，它仅仅是前向渲染路径的一个子集，也就是说，所有可以在顶点照明渲染路径中实现的功能都可以在前向渲染路径中完成。就如它的名字一样，顶点照明渲染路径只是使用了逐顶点的方式来计算光照，并没有什么神奇的地方。实际上，我们在上面的前向渲染路径中也可以计算一些逐顶点的光源。但如果选择使用顶点照明渲染路径，那么Unity会只填充那些逐顶点相关的光源变量，意味着我们不可以使用一些逐像素光照变量。

### 1. Unity中的顶点照明渲染

顶点照明渲染路径通常在一个Pass中就可以完成对物体的渲染。在这个Pass中，我们会计算我们关心的所有光源对该物体的照明，并且这个计算是按逐顶点处理的。这是Unity中最快速的渲染路径，并且具有最广泛的硬件支持（但是游戏机上并不支持这种路径）。

由于顶点照明渲染路径仅仅是前向渲染路径的一个子集，因此在Unity 5发布之前，Unity在论坛上发起了一个投票

（<http://forum.unity3d.com/threads/official-dropping-vertexlit-rendering-path-for-unity-5-0.275248/>），让开发者选择是否应该在Unity 5.0中抛弃顶点照明渲染路径。在这个投票中，很多开发人员表示了赞同的意见。结果是，Unity 5中将顶点照明渲染路径作为一个遗留的渲染路径，在未来的版本中，顶点照明渲染路径的相关设定可能会被移除。

## 2. 可访问的内置变量和函数

在Unity中，我们可以在一个顶点照明的Pass中最多访问到8个逐顶点光源。如果我们只需要渲染其中两个光源对物体的照明，可以仅使用表9.4中内置光照数据的前两个。如果影响该物体的光源数目小于8，那么数组中剩下的光源颜色会设置成黑色。

表9.4 顶点照明渲染路径中可以使用的内置变量

名 称	类 型	描 述
unity_LightColor	half4[8]	光源颜色
unity_LightPosition	float4[8]	xyz分量是视角空间中的光源位置。如果光源是平行光，那么z分量值为0，其他光源类型z分量值为1
unity_LightAtten	half4[8]	光源衰减因子。如果光源是聚光灯，x分量是 $\cos(\text{spotAngle}/2)$ ，y分量是 $1/\cos(\text{spotAngle}/4)$ ；如果是其他类型的光源，x分量是-1，y分量是1。z分量是衰减的平方，w分量是光源范围开根号的结果
unity_SpotDirection	float4[8]	如果光源是聚光灯的话，值为视角空间的聚光灯的位置；如果是其他类型的光源，值为(0, 0, 1, 0)

可以看出，一些变量我们同样可以在前向渲染路径中使用，例如unity\_LightColor。但这些变量数组的维度和数值在不同渲染路径中的值是不同的。

表9.5给出了顶点照明渲染路径中可以使用的内置函数。

表9.5 顶点照明渲染路径中可以使用的内置函数

函 数 名	描 述
float3 ShadeVertexLights (float4 vertex, float3 normal)	输入模型空间中的顶点位置和法线，计算四个逐顶点光源的光照以及环境光。内部实现实际上调用了ShadeVertexLightsFull函数
float3 ShadeVertexLightsFull (float4 vertex, float3 normal, int lightCount, bool spotLight)	输入模型空间中的顶点位置和法线，计算lightCount个光源的光照以及环境光。如果spotLight值为true，那么这些光源会被当成聚光灯来处理，虽然结果更精确，但计算更加耗时；否则，按点光源处理

9.1.3 延迟渲染路径

前向渲染的问题是：当场景中包含大量实时光源时，前向渲染的性能会急速下降。例如，如果我们在场景的某一块区域放置了多个光源，这些光源影响的区域互相重叠，那么为了得到最终的光照效果，我们就需要为该区域内的每个物体执行多个Pass来计算不同光源对该物体的光照结果，然后在颜色缓存中把这些结果混合起来得到最终的光照。然而，每执行一个Pass我们都需要重新渲染一遍物体，但很多计算实际上是重复的。

延迟渲染是一种更古老的渲染方法，但由于上述前向渲染可能造成的瓶颈问题，近几年又流行起来。除了前向渲染中使用的颜色缓冲

和深度缓冲外，延迟渲染还会利用额外的缓冲区，这些缓冲区也被统称为G缓冲（G-buffer），其中G是英文Geometry的缩写。G缓冲区存储了我们所关心的表面（通常指的是离摄像机最近的表面）的其他信息，例如该表面的法线、位置、用于光照计算的材质属性等。

## 1. 延迟渲染的原理

延迟渲染主要包含了两个Pass。在第一个Pass中，我们不进行任何光照计算，而是仅仅计算哪些片元是可见的，这主要是通过深度缓冲技术来实现，当发现一个片元是可见的，我们就把它的相关信息存储到G缓冲区中。然后，在第二个Pass中，我们利用G缓冲区的各个片元信息，例如表面法线、视角方向、漫反射系数等，进行真正的光照计算。

延迟渲染的过程大致可以用下面的伪代码来描述：

```
Pass 1 {
    // 第一个Pass不进行真正的光照计算
    // 仅仅把光照计算需要的信息存储到G缓冲中

    for (each primitive in this model) {
        for (each fragment covered by this primitive) {
            if (failed in depth test) {
                // 如果没有通过深度测试，说明该片元是不可见的
                discard;
            } else {
                // 如果该片元可见
                // 就把需要的信息存储到G缓冲中
                writeGBuffer(materialInfo, pos, normal);
            }
        }
    }
}

Pass 2 {
    // 利用G缓冲中的信息进行真正的光照计算

    for (each pixel in the screen) {
```

```
    if (the pixel is valid) {  
        // 如果该像素是有效的  
        // 读取它对应的G缓冲中的信息  
        readGBuffer(pixel, materialInfo, pos, normal);  
  
        // 根据读取到的信息进行光照计算  
        float4 color = Shading(materialInfo, pos, normal,  
lightDir, viewDir);  
        // 更新帧缓冲  
        writeFrameBuffer(pixel, color);  
    }  
}  
}
```

可以看出，延迟渲染使用的Pass数目通常就是两个，这跟场景中包含的光源数目是没有关系的。换句话说，延迟渲染的效率不依赖于场景的复杂度，而是和我们使用的屏幕空间的大小有关。这是因为，我们需要的信息都存储在缓冲区中，而这些缓冲区可以理解成是一张张2D图像，我们的计算实际上就是在这些图像空间中进行的。

## 2. Unity中的延迟渲染

Unity有两种延迟渲染路径，一种是遗留的延迟渲染路径，即Unity 5之前使用的延迟渲染路径，而另一种是Unity 5.x中使用的延迟渲染路径。如果游戏中使用了大量的实时光照，那么我们可能希望选择延迟渲染路径，但这种路径需要一定的硬件支持。

新旧延迟渲染路径之间的差别很小，只是使用了不同的技术来权衡不同的需求。例如，较旧版本的延迟渲染路径不支持Unity 5的基于物理的Standard Shader。以下我们仅讨论Unity 5后使用的延迟渲染路径。对于遗留的延迟渲染路径，读者可以在官方文档

（<http://docs.unity3d.com/Manual/RenderTech-DeferredLighting.html>）找到更多的资料。

对于延迟渲染路径来说，它最适合在场景中光源数目很多、如果使用前向渲染会造成性能瓶颈的情况下使用。而且，延迟渲染路径中的每个光源都可以按逐像素的方式处理。但是，延迟渲染也有一些缺点。

- 不支持真正的抗锯齿（**anti-aliasing**）功能。
- 不能处理半透明物体。
- 对显卡有一定要求。如果要使用延迟渲染的话，显卡必须支持 **MRT**（**Multiple Render Targets**）、**Shader Mode 3.0**及以上、深度渲染纹理以及双面的模板缓冲。

当使用延迟渲染时，**Unity**要求我们提供两个**Pass**。

（1）第一个**Pass**用于渲染**G**缓冲。在这个**Pass**中，我们会把物体的漫反射颜色、高光反射颜色、平滑度、法线、自发光和深度等信息渲染到屏幕空间的**G**缓冲区中。对于每个物体来说，这个**Pass**仅会执行一次。

（2）第二个**Pass**用于计算真正的光照模型。这个**Pass**会使用上一个**Pass**中渲染的数据来计算最终的光照颜色，再存储到帧缓冲中。

默认的**G**缓冲区（注意，不同**Unity**版本的渲染纹理存储内容会有所不同）包含了以下几个渲染纹理（**Render Texture**，**RT**）。

- **RT0**：格式是**ARGB32**，**RGB**通道用于存储漫反射颜色，**A**通道没有被使用。
- **RT1**：格式是**ARGB32**，**RGB**通道用于存储高光反射颜色，**A**通道用于存储高光反射的指数部分。

- RT2: 格式是ARGB2101010, RGB通道用于存储法线, A通道没有被使用。
- RT3: 格式是ARGB32 (非HDR) 或ARGBHalf (HDR), 用于存储自发光+lightmap+反射探针 (reflection probes)。
- 深度缓冲和模板缓冲。

当在第二个Pass中计算光照时, 默认情况下仅可以使用Unity内置的Standard光照模型。如果我们想要使用其他的光照模型, 就需要替换掉原有的Internal-DeferredShading.shader文件。更详细的信息可以访问官方文档 (<http://docs.unity3d.com/Manual/RenderTech-DeferredShading.html>)。

### 3. 可访问的内置变量和函数

表 9.6 给出了处理延迟渲染路径可以使用的光照变量。这些变量都可以在UnityDeferred Library.cginc文件中找到它们的声明。

表9.6 延迟渲染路径中可以使用的内置变量

名 称	类 型	描 述
_LightColor	float4	光源颜色
_LightMatrix0	float4×4	从世界空间到光源空间的变换矩阵。可以用于采样cookie和光强衰减纹理

#### 9.1.4 选择哪种渲染路径



Unity的官方文档

(<http://docs.unity3d.com/Manual/RenderingPaths.html>) 中给出了4种渲染路径（前向渲染路径、延迟渲染路径、遗留的延迟渲染路径和顶点照明渲染路径）的详细比较，包括它们的特性比较（是否支持逐像素光照、半透明物体、实时阴影等）、性能比较以及平台支持。

总体来说，我们需要根据游戏发布的目标平台来选择渲染路径。如果当前显卡不支持所选渲染路径，那么Unity会自动使用比其低一级的渲染路径。

在本书中，我们主要使用Unity的前向渲染路径。

## 9.2 Unity的光源类型

在前面的例子中，我们的场景中都仅仅有一个光源且光源类型是平行光（如果你的场景不是这样的话，可能会得到错误的结果）。只有一个平行光的世界很美好，但美梦总有醒的一天，这时，我们就需要在Unity Shader中处理更复杂的光源类型以及数目更多的光源。在本节中，我们将会学习如何在Unity中处理**点光源（point light）**和**聚光灯（spot light）**。

Unity一共支持4种光源类型：平行光、点光源、聚光灯和**面光源（area light）**。面光源仅在烘焙时才可发挥作用，因此不在本节讨论范围内。由于每种光源的几何定义不同，因此它们对应的光源属性也就各不相同。这就要求我们要区别对待它们。幸运的是，Unity提供了很多内置函数来帮我们处理这些光源，在本章的最后我们会介绍这些函数，但首先我们需要了解它们背后的原理。

## 9.2.1 光源类型有什么影响

我们来看一下光源类型的不同到底会给Shader带来哪些影响。我们可以考虑Shader中使用了光源的哪些属性。最常使用的光源属性有光源的**位置**、**方向**（更具体说就是，到某点的方向）、**颜色**、**强度**以及**衰减**（更具体说就是，到某点的衰减，与该点到光源的距离有关）这5个属性。而这些属性和它们的几何定义息息相关。

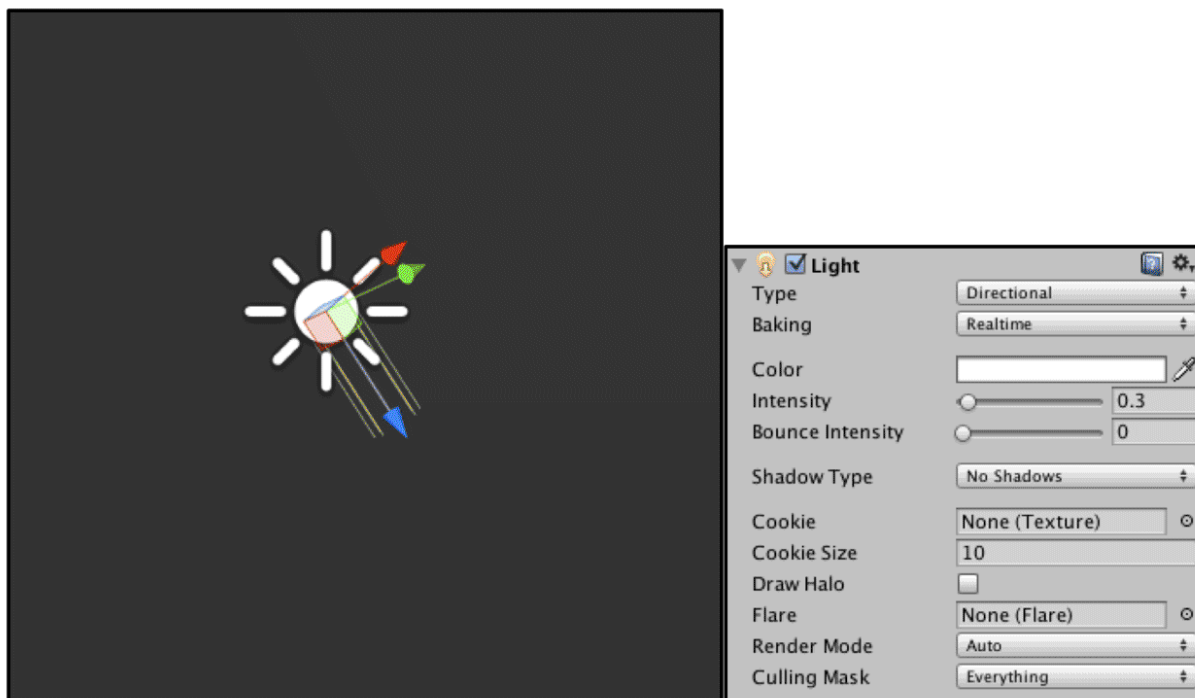
### 1. 平行光

对于我们之前使用的平行光来说，它的几何定义是最简单的。平行光可以照亮的范围是没有限制的，它通常是作为太阳这样的角色在场景中出现的。图9.5给出了Unity中平行光在Scene视图中的表示以及Light组件的面板。

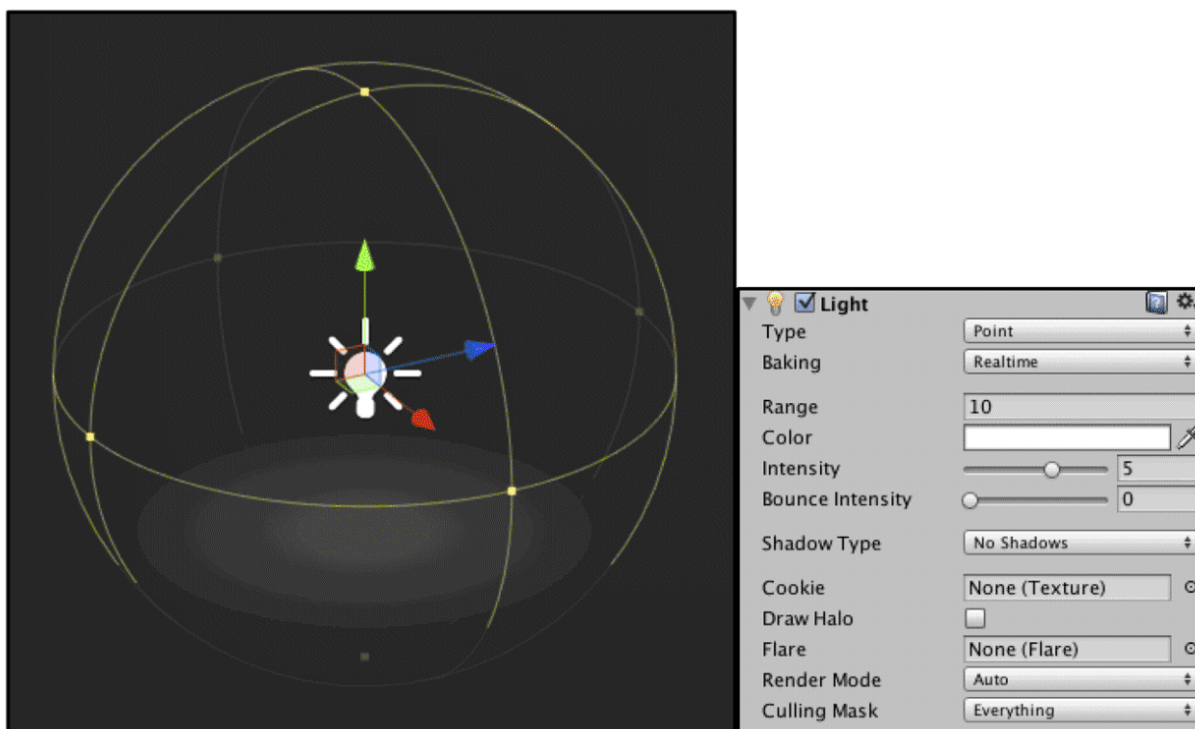
平行光之所以简单，是因为它没有一个唯一的位置，也就是说，它可以放在场景中的任意位置（回忆一下，我们小时候是不是总感觉太阳跟着我们一起移动）。它的几何属性只有方向，我们可以调整平行光的Transform组件中的Rotation属性来改变它的光源方向，而且平行光到场景中所有点的方向都是一样的，这也是平行光名字的由来。除此之外，由于平行光没有一个具体的位置，因此也没有衰减的概念，也就是说，光照强度不会随着距离而发生改变。

### 2. 点光源

点光源的照亮空间则是有限的，它是由空间中的一个球体定义的。点光源可以表示由一个点发出的、向所有方向延伸的光。图9.6给出了Unity中点光源在Scene视图中的表示以及Light组件的面板。



▲图9.5 平行光



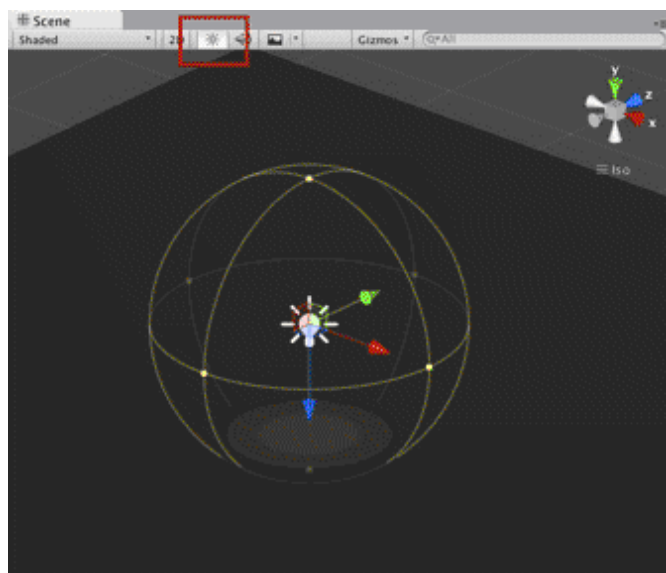
▲图9.6 点光源

需要提醒读者的一点是，我们需要在Scene视图中开启光照才能看到预览光源是如何影响场景中的物体的。图9.7给出了开启Scene视图光照的按钮。

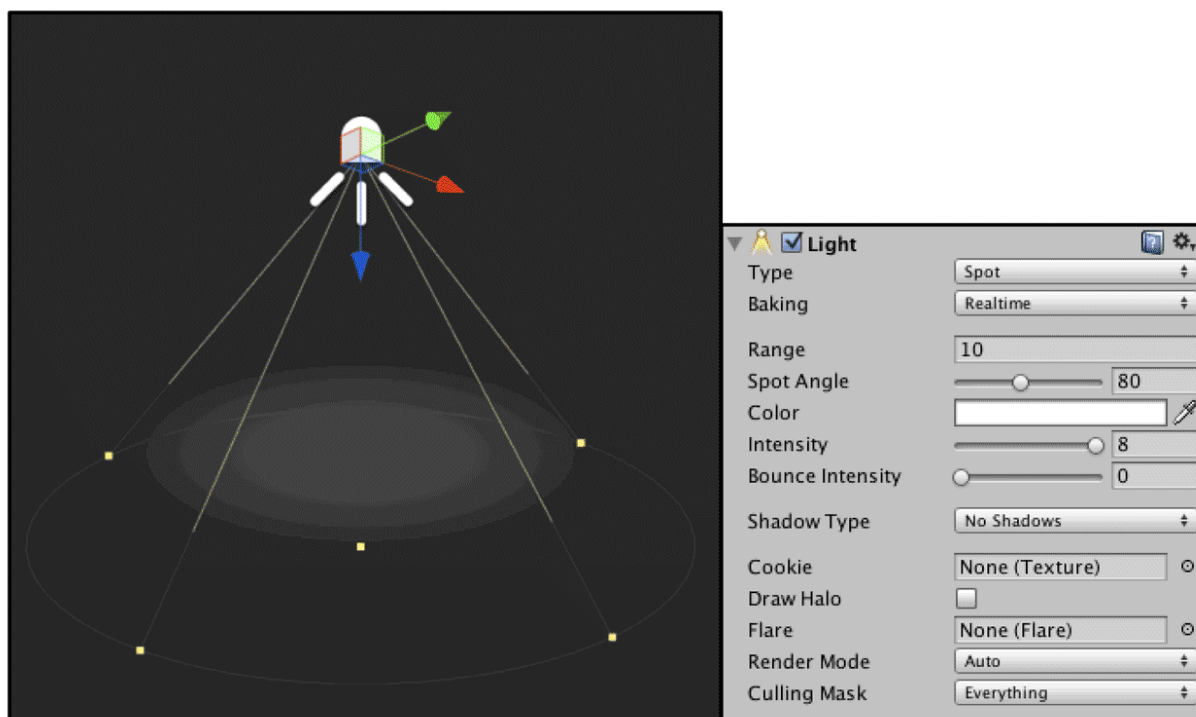
球体的半径可以由面板中的Range属性来调整，也可以在Scene视图中直接拖拉点光源的线框（如球体上的黄色控制点）来修改它的属性。点光源是有位置属性的，它是由点光源的Transform组件中的Position属性定义的。对于方向属性，我们需要用点光源的位置减去某点的位置来得到它到该点的方向。而点光源的颜色和强度可以在Light组件面板中调整。同时，点光源也是会衰减的，随着物体逐渐远离点光源，它接收到的光照强度也会逐渐减小。点光源球心处的光照强度最强，球体边界处的最弱，值为0。其中间的衰减值可以由一个函数定义。

### 3. 聚光灯

聚光灯是这3种光源类型中最复杂的一种。它的照亮空间同样是有限的，但不再是简单的球体，而是由空间中一块锥形区域定义的。聚光灯可以用于表示由一个特定位置出发、向特定方向延伸的光。图9.8给出了Unity中聚光灯在Scene视图中的表示以及Light组件的面板。



▲ 图9.7 开启Scene视图中的光照



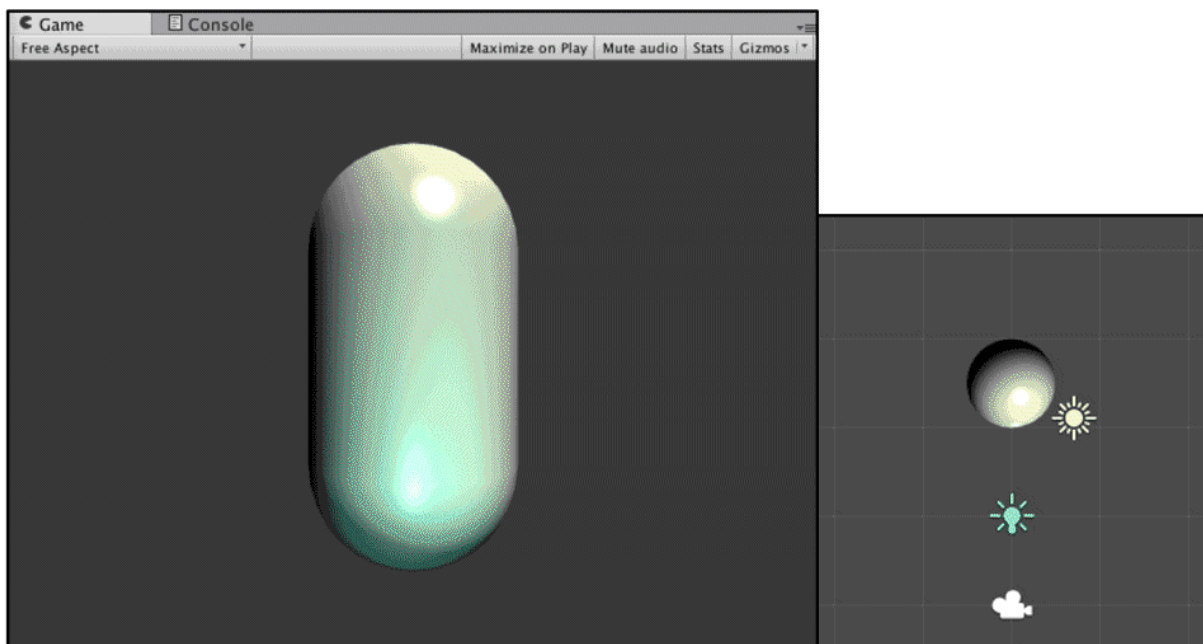
▲ 图9.8 聚光灯

这块锥形区域的半径由面板中的**Range**属性决定，而锥体的张开角度由**Spot Angle**属性决定。我们同样也可以在**Scene**视图中直接拖拉聚光灯的线框（如中间的黄色控制点以及四周的黄色控制点）来修改它的属性。聚光灯的位置同样是由**Transform**组件中的**Position**属性定义的。对于方向属性，我们需要用聚光灯的位置减去某点的位置来得到它到该点的方向。聚光灯的衰减也是随着物体逐渐远离点光源而逐渐减小，在锥形的顶点处光照强度最强，在锥形的边界处强度为0。中间的衰减值可以由一个函数定义，这个函数相对于点光源衰减计算公式要更加复杂，因为我们需要判断一个点是否在锥体的范围内。

### 9.2.2 在前向渲染中处理不同的光源类型

在了解了3种光源的几何定义后，我们来看一下如何在Unity Shader中访问它们的5个属性：**位置、方向、颜色、强度以及衰减**。需要注意的是，本节均建立在使用前向渲染路径的基础上。

在学习完本节后，我们可以得到类似图9.9中的效果。



▲图9.9 使用一个平行光和一个点光源共同照亮物体。右图显示了胶囊体、平行光和点光源在场景中的相对位置

## 1. 实践

为了实现上述效果，我们首先做如下准备工作。

(1) 在Unity中新建一个场景。在本书资源中，该场景名为Scene\_9\_2\_2\_1。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window → Lighting → Skybox中去掉场景中的天空盒子。

(2) 新建一个材质。在本书资源中，该材质名为ForwardRenderingMat。

(3) 新建一个Unity Shader。在本书资源中，该Shader名为Chapter9-ForwardRendering。把新的Unity Shader赋给第2步中创建的材



质。

(4) 在场景中创建一个胶囊体，并把第2步中的材质赋给该胶囊体。

(5) 为了让物体受多个光源的影响，我们再新建一个点光源，将其颜色设为绿色，以和平行光进行区分。

(6) 保存场景。

我们的代码使用了Blinn-Phong光照模型，并为前向渲染定义了Base Pass和Additional Pass来处理多个光源。在这里我们只给出其中关键的代码，而省略与之前章节中重复的代码。完整的代码读者可以在本书资源中找到。关键代码如下。

(1) 我们首先定义第一个Pass——Base Pass。为此，我们需要设置该Pass的渲染路径标签：

```
Pass {  
    // Pass for ambient light & first pixel light (directional  
    light)  
    Tags { "LightMode"="ForwardBase" }  
  
    CGPROGRAM  
  
    // Apparently need to add this declaration  
    #pragma multi_compile_fwdbase
```

需要注意的是，我们除了设置渲染路径外，还使用了**#pragma**编译指令。**#pragma multi\_compile\_fwdbase**指令可以保证我们在Shader中使用光照衰减等光照变量可以被正确赋值。这是不可缺少的。

(2) 在Base Pass的片元着色器中，我们首先计算了场景中的环境光：

```
// Get ambient term
fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;
```

我们希望环境光计算一次即可，因此在后面的Additional Pass中就不会再计算这个部分。与之类似，还有物体的自发光，但在本例中，我们假设胶囊体没有自发光效果。

(3) 然后，我们在Base Pass中处理了场景中的最重要的平行光。在这个例子中，场景中只有一个平行光。如果场景中包含了多个平行光，Unity会选择最亮的平行光传递给Base Pass进行逐像素处理，其他平行光会按照逐顶点或在Additional Pass中按逐像素的方式处理。如果场景中没有任何平行光，那么Base Pass会当成全黑的光源处理。我们提到过，每一个光源有5个属性：**位置、方向、颜色、强度**以及**衰减**。对于Base Pass来说，它处理的逐像素光源类型一定是平行光。我们可以使用\_worldSpaceLightPos0来得到这个平行光的方向（位置对平行光来说没有意义），使用\_LightColor0来得到它的颜色和强度

（\_LightColor0已经是颜色和强度相乘后的结果），由于平行光可以认为是没有衰减的，因此这里我们直接令衰减值为1.0。相关代码如下：

```
// Compute diffuse term
fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb * max(0,
dot(worldNormal, worldLightDir));

...

// Compute specular term
fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0,
dot(worldNormal, halfDir)), _Gloss);

// The attenuation of directional light is always 1
```

```
fixed atten = 1.0;

return fixed4(ambient + (diffuse + specular) * atten, 1.0);
```

至此，**Base Pass**的工作就完成了。

(4) 接下来，我们需要为场景中其他逐像素光源定义**Additional Pass**。为此，我们首先需要设置**Pass**的渲染路径标签：

```
Pass {
    // Pass for other pixel lights
    Tags { "LightMode"="ForwardAdd" }

    Blend One One

    CGPROGRAM

    // Apparently need to add this declaration
    #pragma multi_compile_fwdadd
```

除了设置渲染路径标签外，我们同样使用了**#pragma multi\_compile\_fwdadd**指令，如前面所说，这个指令可以保证我们在**Additional Pass**中访问到正确的光照变量。与**Base Pass**不同的是，我们还使用**Blend**命令开启和设置了混合模式。这是因为，我们希望**Additional Pass**计算得到的光照结果可以在帧缓存中与之前的光照结果进行叠加。如果没有使用**Blend**命令的话，**Additional Pass**会直接覆盖掉之前的光照结果。在本例中，我们选择的混合系数是**Blend One One**，这不是必需的，我们可以设置成Unity支持的任何混合系数。常见的还有**Blend SrcAlpha One**。

(5) 通常来说，**Additional Pass**的光照处理和**Base Pass**的处理方式是一样的，因此我们只需要把**Base Pass**的顶点和片元着色器代码粘贴到**Additional Pass**中，然后再稍微修改一下即可。这些修改往往是为了

去掉Base Pass中环境光、自发光、逐顶点光照、SH光照的部分，并添加一些对不同光源类型的支持。因此，在Additional Pass的片元着色器中，我们没有再计算场景中的环境光。由于Additional Pass处理的光源类型可能是平行光、点光源或是聚光灯，因此在计算光源的5个属性——**位置、方向、颜色、强度**以及**衰减**时，颜色和强度我们仍然可以使用\_LightColor0来得到，但对于位置、方向和衰减属性，我们就需要根据光源类型分别计算。首先，我们来看如何计算不同光源的方向：

```
#ifdef USING_DIRECTIONAL_LIGHT
    fixed3 worldLightDir = normalize(_WorldSpaceLightPos0.xyz);
#else
    fixed3 worldLightDir = normalize(_WorldSpaceLightPos0.xyz -
i.worldPosition.xyz);
#endif
```

在上面的代码中，我们首先判断了当前处理的逐像素光源的类型，这是通过使用#ifdef指令判断是否定义了USING\_DIRECTIONAL\_LIGHT来得到的。如果当前前向渲染Pass处理的光源类型是平行光，那么Unity的底层渲染引擎就会定义USING\_DIRECTIONAL\_LIGHT。如果判断得知是平行光的话，光源方向可以直接由\_WorldSpaceLightPos0.xyz得到；如果是点光源或聚光灯，那么\_WorldSpaceLightPos0.xyz表示的是世界空间下的光源位置，而想要得到光源方向的话，我们就需要用这个位置减去世界空间下的顶点位置。

(6) 最后，我们需要处理不同光源的衰减：

```
#ifdef USING_DIRECTIONAL_LIGHT
    fixed atten = 1.0;
#else
    float3 lightCoord = mul(_LightMatrix0, float4(i.worldPosition,
1)).xyz;
```

```
fixed atten = tex2D(_LightTexture0, dot(lightCoord,  
lightCoord).rr).UNITY_ATTEN_CHANNEL;  
#endif
```

我们同样通过判断是否定义了USING\_DIRECTIONAL\_LIGHT来决定当前处理的光源类型。如果是平行光的话，衰减值为1.0。如果是其他光源类型，那么处理更复杂一些。尽管我们可以使用数学表达式来计算给定点相对于点光源和聚光灯的衰减，但这些计算往往涉及开根号、除法等计算量相对较大的操作，因此Unity选择了使用一张纹理作为查找表（Lookup Table，LUT），以在片元着色器中得到光源的衰减。我们首先得到光源空间下的坐标，然后使用该坐标对衰减纹理进行采样得到衰减值。关于Unity中衰减纹理的细节可以参见9.3节。

我们可以在场景中添加更多的逐像素光源来照亮胶囊体。**需要注意的是**，本节只是为了讲解处理其他类型光源的实现原理，上述代码并不会用于真正的项目中，我们会在9.5节给出包含了完整光照计算的Unity Shader。

## 2. 实验：Base Pass和Additional Pass的调用

我们在9.1.1节中给出了前向渲染中Unity是如何决定哪些光源是逐像素光，而哪些是逐顶点或SH光。为了让读者有更加直观的理解，我们可以在Unity中进行一个实验。实验的准备工作如下。

(1) 在Unity中新建一个场景。在本书资源中，该场景名为Scene\_9\_2\_2\_2。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window -> Lighting -> Skybox中去掉场景中的天空盒子。

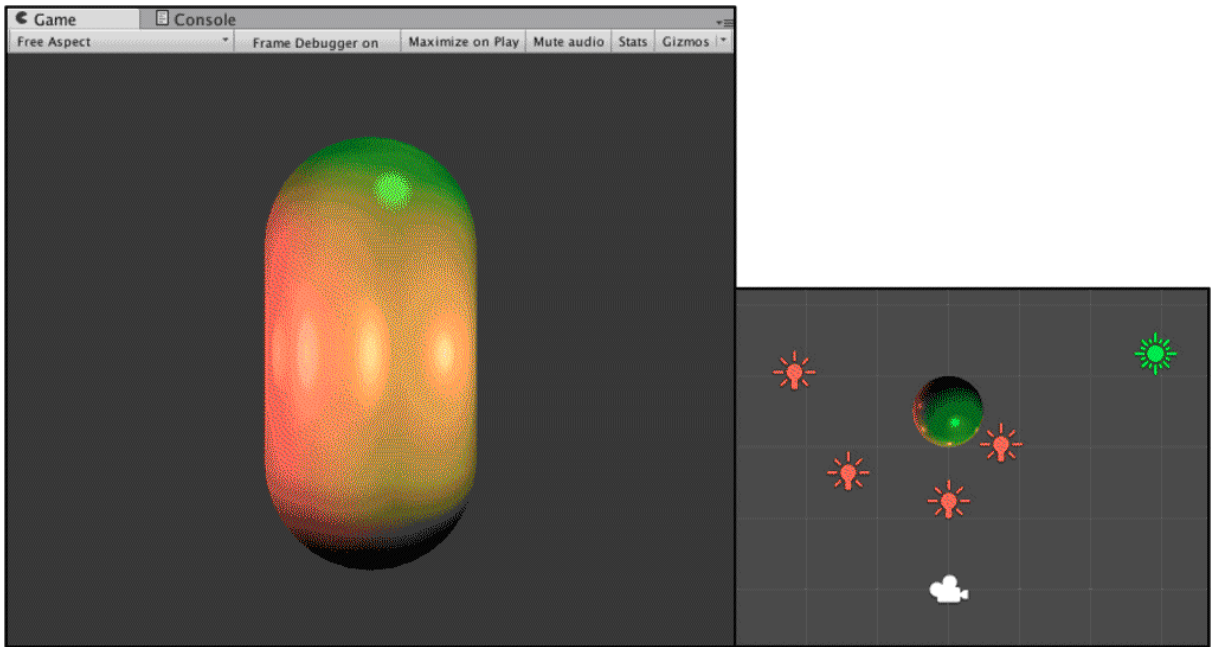
(2) 调整平行光的颜色为绿色。

(3) 在场景中创建一个胶囊体，并把上一节中的ForwardRenderingMat材质赋给该胶囊体。

(4) 新建4个点光源，调整它们的颜色为相同的红色。

(5) 保存场景。

我们可以得到类似图9.10中的效果。



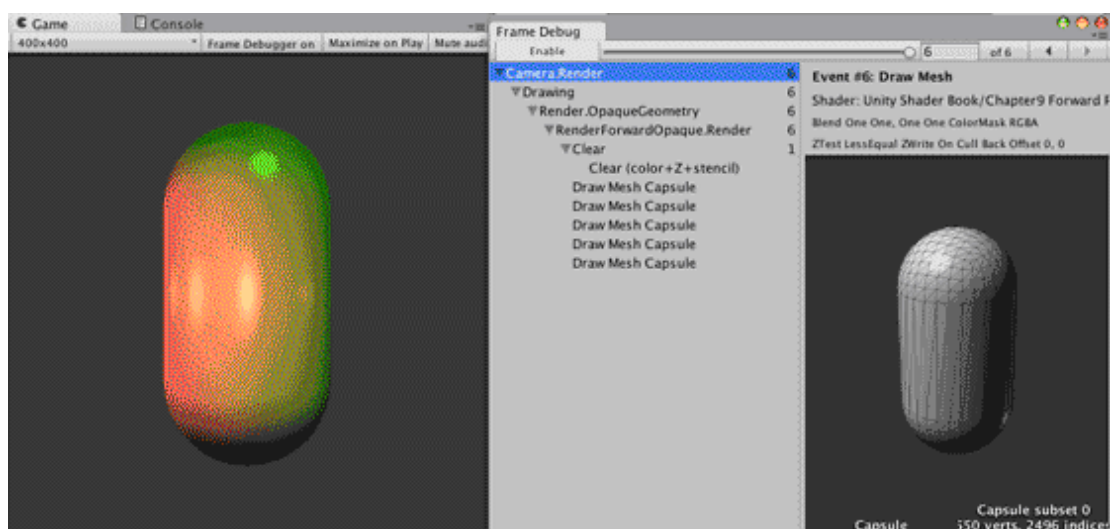
▲ 图9.10 使用1个平行光 + 4个点光源照亮一个物体

那么，这样的结果是怎么来的呢？当我们创建一个光源时，默认情况下它的**Render Mode**（可以在Light组件中设置）是**Auto**。这意味着，Unity会在背后为我们判断哪些光源会按逐像素处理，而哪些按逐顶点或SH的方式处理。由于我们没有更改Edit → Project Settings →

Quality → Pixel Light Count中的数值，因此默认情况下一个物体可以接收除最亮的平行光外的4个逐像素光照。在这个例子中，场景中共包含了5个光源，其中一个为平行光，它会在Chapter9-Forward Rendering的Base Pass中按逐像素的方式被处理；其余4个都是点光源，由于它们的Render Mode为Auto且数目正好等于4，因此都会在Chapter9-ForwardRendering的Additional Pass中逐像素的方式被处理，每个光源会调用一次Additional Pass。

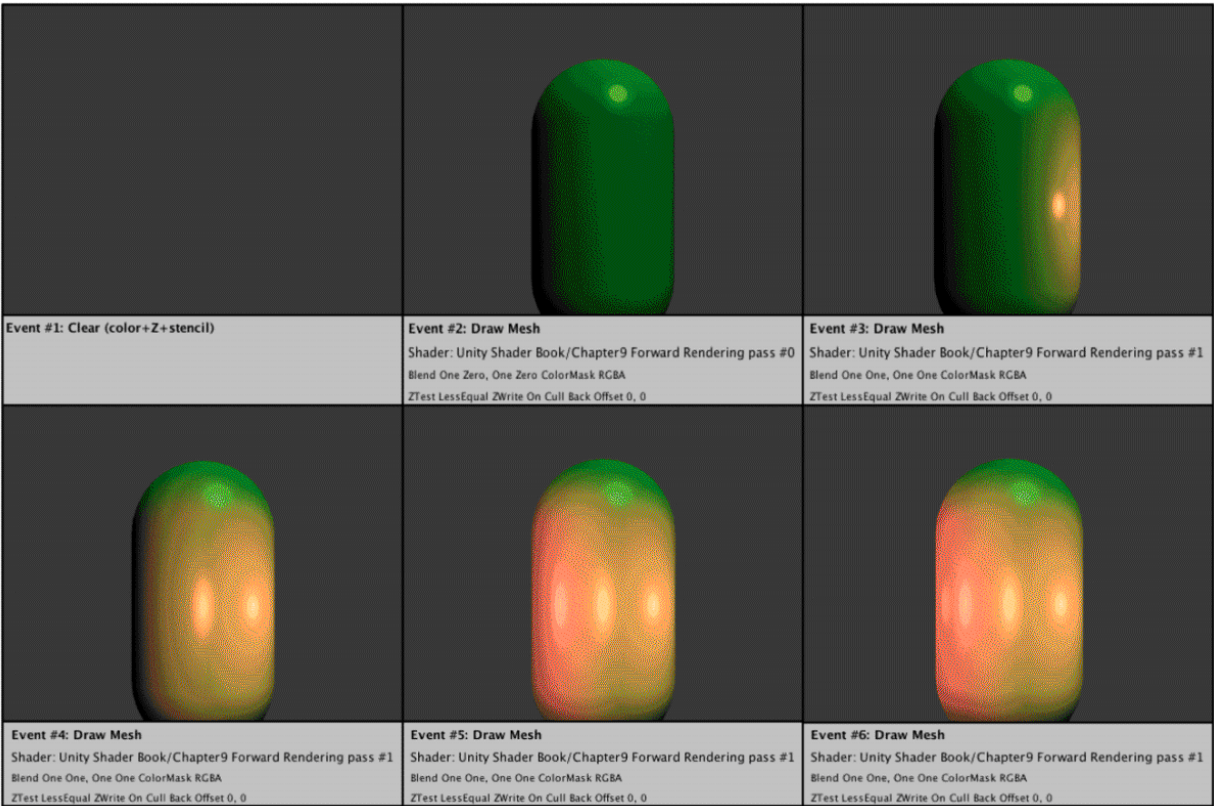
在Unity 5中，我们还可以使用**帧调试器（Frame Debugger）**工具来查看场景的绘制过程。使用方法是：在Window -> Frame Debugger中打开帧调试器，如图9.11所示。

从帧调试器中可以看出，渲染这个场景Unity一共进行了6个渲染事件，由于本例中只包含了一个物体，因此这6个渲染事件几乎都是用于渲染该物体的光照结果。我们可以通过依次单击帧调试器中的渲染事件，来查看Unity是怎样渲染物体的。图9.12给出了本例中Unity进行的6个渲染事件。





▲ 图9.11 打开帧调试器查看场景的绘制事件



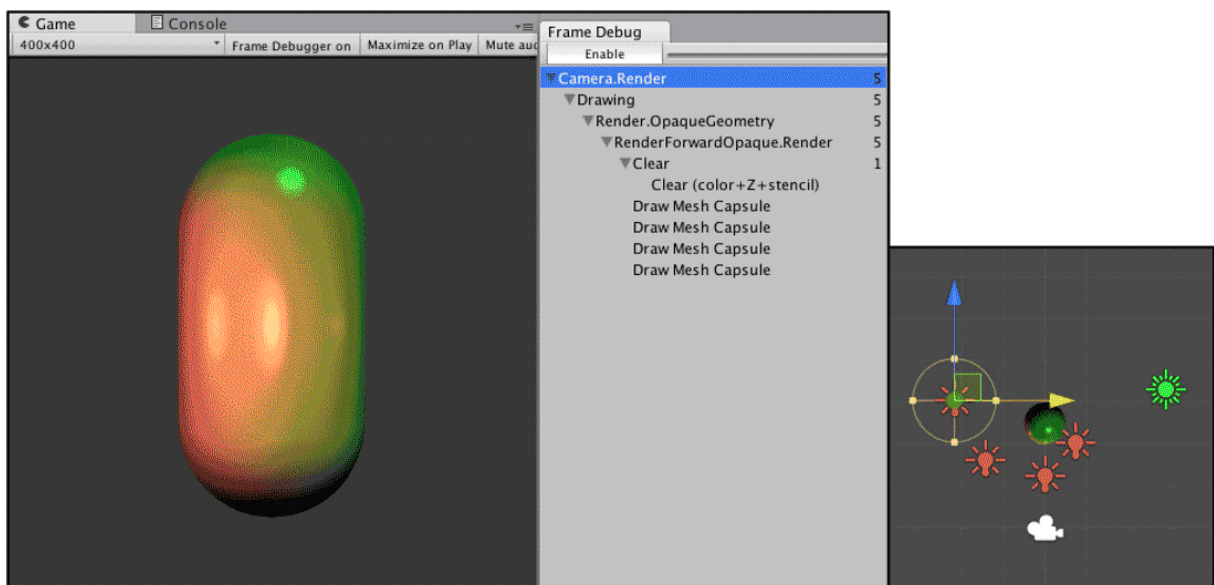
▲ 图9.12 本例中的6个渲染事件，绘制顺序是从左到右、从上到下进行的

从图9.12可以看出，Unity是如何一步步将不同光照渲染到物体上的：在第一个渲染事件中，Unity首先清除颜色、深度和模板缓冲，为后面的渲染做准备；在第二个渲染事件中，Unity利用Chapter9-ForwardRendering的第一个Pass，即Base Pass，将平行光的光照渲染到帧缓存中；在后面的4个渲染事件中，Unity使用Chapter9-ForwardRendering的第二个Pass，即Additional Pass，依次将4个点光源的光照应用到物体上，得到最后的渲染结果。

可以注意到，Unity处理这些点光源的顺序是按照它们的重要度排序的。在这个例子中，由于所有点光源的颜色和强度都相同，因此它

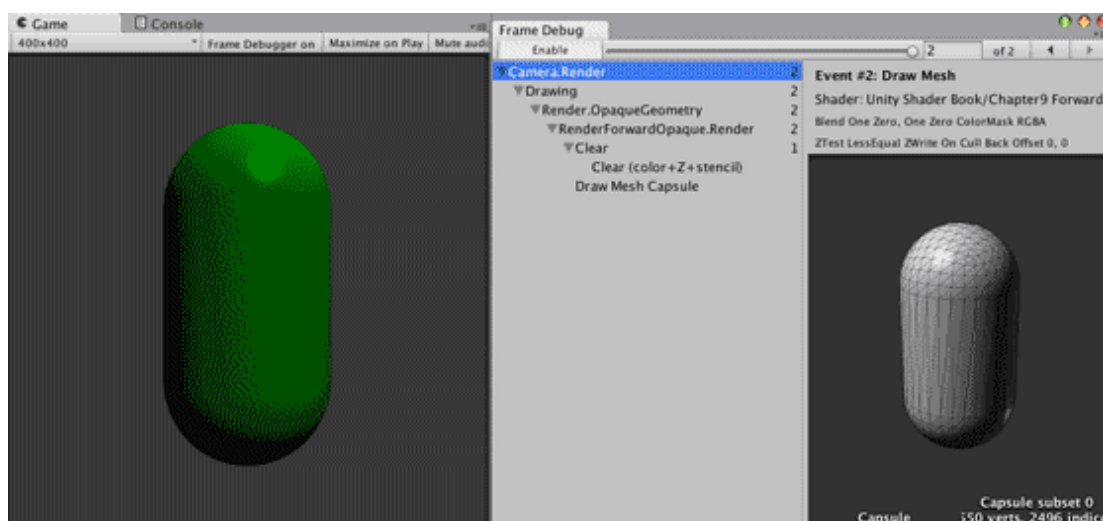
们的重要度取决于它们距离胶囊体的远近，因此图9.12中首先绘制的是距离胶囊体最近的点光源。但是，如果光源的强度和颜色互不相同，那么距离就不再是唯一的衡量标准。例如，如果我们把现在距离最近的点光源的强度设为0.2，那么从帧调试器中我们可以发现绘制顺序发生了变化，此时首先绘制的是距离胶囊体第二近的点光源，最近的点光源则会在最后被渲染。Unity官方文档中并没有给出光源强度、颜色和距离物体的远近是如何具体影响光源的重要度排序的，我们仅知道排序结果和这三者都有关系。

对于场景中的一个物体，如果它不在一个光源的光照范围内，Unity是不会为这个物体调用Pass来处理这个光源的。我们可以把本例中距离最远的点光源的范围调小，使得胶囊体在它的照亮范围外。此时再查看帧调试器，我们可以发现渲染事件比之前少了一个，如图9.13所示。同样，如果一个物体不在某个聚光灯的范围内，Unity也是不会为该物体调用相关的渲染事件的。



▲图9.13 如果物体不在一个光源的光照范围内（从右图可以看出，胶囊体不在最左方的点光源的照明范围内），Unity是不会调用Additional Pass来为该物体处理该光源的

我们知道，如果逐像素光源的数目很多的话，该物体的Additional Pass就会被调用多次，影响性能。我们可以通过把光源的**Render Mode**设为**Not Important**来告诉Unity，我们不希望把该光源当成逐像素处理。在本例中，我们可以把4个点光源的**Render Mode**都设为**Not Important**，可以得到图9.14中的结果。



▲图9.14 当把光源的**Render Mode**设为**Not Important**时，这些光源就不会按逐像素光来处理

由于我们在Chapter9-ForwardRendering中没有在Base Pass中计算逐顶点和SH光源，因此场景中的4个点光源实际上不会对物体造成任何影响。同样，如果我们把平行光的**Render Mode**也设为**Not Important**，那么读者可以猜测一下结果会是什么。没错，物体就会仅显示环境光的光照结果。

那么，我们如何在前向渲染路径的Base Pass中计算逐顶点和SH光呢？我们可以使用9.1.1节中提到的内置变量和函数来计算这些光源的

光照效果。

## 9.3 Unity的光照衰减

在9.2节中，我们提到Unity使用一张纹理作为查找表来在片元着色器中计算逐像素光照的衰减。这样的好处在于，计算衰减不依赖于数学公式的复杂性，我们只要使用一个参数值去纹理中采样即可。但使用纹理查找来计算衰减也有一些弊端。

- 需要预处理得到采样纹理，而且纹理的大小也会影响衰减的精度。
- 不直观，同时也不方便，因此一旦把数据存储在查找表中，我们就无法使用其他数学公式来计算衰减。

但由于这种方法可以在一定程度上提升性能，而且得到的效果在大部分情况下都是良好的，因此Unity默认就是使用这种纹理查找的方式来计算逐像素的点光源和聚光灯的衰减的。

### 9.3.1 用于光照衰减的纹理

Unity在内部使用一张名为\_LightTexture0的纹理来计算光源衰减。需要注意的是，如果我们对光源使用了cookie，那么衰减查找纹理是\_LightTextureB0，但这里不讨论这种情况。我们通常只关心\_LightTexture0对角线上的纹理颜色值，这些值表明了光源空间中不同位置的点的衰减值。例如，(0, 0)点表明了与光源位置重合的点的衰减值，而(1, 1)点表明了光源空间中所关心的距离最远的点的衰减。

为了对`_LightTexture0`纹理采样得到给定点到该光源的衰减值，我们首先需要得到该点在光源空间中的位置，这是通过`_LightMatrix0`变换矩阵得到的。在9.1.1节中，我们已经知道`_LightMatrix0`可以把顶点从世界空间变换到光源空间。因此，我们只需要把`_LightMatrix0`和世界空间中的顶点坐标相乘即可得到光源空间中的相应位置：

```
float3 lightCoord = mul(_LightMatrix0, float4(i.worldPosition, 1)).xyz;
```

然后，我们可以使用这个坐标的模的平方对衰减纹理进行采样，得到衰减值：

```
fixed atten = tex2D(_LightTexture0, dot(lightCoord, lightCoord).rr).UNITY_ATTEN_CHANNEL;
```

可以发现，在上面的代码中，我们使用了光源空间中顶点距离的平方（通过`dot`函数来得到）来对纹理采样，之所以没有使用距离值来采样是因为这种方法可以避免开方操作。然后，我们使用宏`UNITY_ATTEN_CHANNEL`来得到衰减纹理中衰减值所在的分量，以得到最终的衰减值。

### 9.3.2 使用数学公式计算衰减

尽管纹理采样的方法可以减少计算衰减时的复杂度，但有时我们希望可以在代码中利用公式来计算光源的衰减。例如，下面的代码可以计算光源的线性衰减：

```
float distance = length(_WorldSpaceLightPos0.xyz - i.worldPosition.xyz);  
atten = 1.0 / distance; // linear attenuation
```



可惜的是，Unity没有在文档中给出内置衰减计算的相关说明。尽管我们仍然可以在片元着色器中利用一些数学公式来计算衰减，但由于我们无法在Shader中通过内置变量得到光源的范围、聚光灯的朝向、张开角度等信息，因此得到的效果往往在有些时候不尽如人意，尤其在物体离开光源的照明范围时会发生突变（这是因为，如果物体不在该光源的照明范围内，Unity就不会为物体执行一个Additional Pass）。当然，我们可以利用脚本将光源的相关信息传递给Shader，但这样的灵活性很低。我们只能期待未来的版本中Unity可以完善文档并开放更多的参数给开发者使用。

## 9.4 Unity的阴影

为了让场景看起来更加真实，具有深度信息，我们通常希望光源可以把一些物体的阴影投射在其他物体上。在本节，我们就来学习如何在Unity中让一个物体向其他物体投射阴影，以及如何让一个物体接收来自其他物体的阴影。

### 9.4.1 阴影是如何实现的

我们可以先考虑真实生活中阴影是如何产生的。当一个光源发射的一条光线遇到一个不透明物体时，这条光线就不可以再继续照亮其他物体（这里不考虑光线反射）。因此，这个物体就会向它旁边的物体投射阴影，那些阴影区域的产生是因为光线无法到达这些区域。

在实时渲染中，我们最常使用的是一种名为**Shadow Map**的技术。这种技术理解起来非常简单，它会首先把摄像机的位置放在与光源重

合的位置上，那么场景中该光源的阴影区域就是那些摄像机看不到的地方。而Unity就是使用的这种技术。

在前向渲染路径中，如果场景中最重要平行光开启了阴影，Unity就会为该光源计算它的阴影映射纹理（shadowmap）。这张阴影映射纹理本质上也是一张深度图，它记录了从该光源的位置出发、能看到的场景中距离它最近的表面位置（深度信息）。

那么，在计算阴影映射纹理时，我们如何判定距离它最近的表面位置呢？一种方法是，先把摄像机放置到光源的位置上，然后按正常的渲染流程，即调用Base Pass和Additional Pass来更新深度信息，得到阴影映射纹理。但这种方法会对性能造成一定的浪费，因为我们实际上仅仅需要深度信息而已，而Base Pass和Additional Pass中往往涉及很多复杂的光照模型计算。因此，Unity选择使用一个额外的Pass来专门更新光源的阴影映射纹理，这个Pass就是**LightMode**标签被设置为**ShadowCaster**的Pass。这个Pass的渲染目标不是帧缓存，而是阴影映射纹理（或深度纹理）。Unity首先把摄像机放置到光源的位置上，然后调用该Pass，通过对顶点变换后得到光源空间下的位置，并据此来输出深度信息到阴影映射纹理中。因此，当开启了光源的阴影效果后，底层渲染引擎首先会在当前渲染物体的Unity Shader中找到**LightMode**为**ShadowCaster**的Pass，如果没有，它就会在**Fallback**指定的Unity Shader中继续寻找，如果仍然没有找到，该物体就无法向其他物体投射阴影（但它仍然可以接收来自其他物体的阴影）。当找到了一个**LightMode**为**ShadowCaster**的Pass后，Unity会使用该Pass来更新光源的阴影映射纹理。



在传统的阴影映射纹理的实现中，我们会在正常渲染的Pass中把顶点位置变换到光源空间下，以得到它在光源空间中的三维位置信息。然后，我们使用xy分量对阴影映射纹理进行采样，得到阴影映射纹理中该位置的深度信息。如果该深度值小于该顶点的深度值（通常由z分量得到），那么说明该点位于阴影中。但在Unity 5中，Unity使用了不同于这种传统的阴影采样技术，即**屏幕空间的阴影映射技术**

（**Screenspace Shadow Map**）。屏幕空间的阴影映射原本是延迟渲染中产生阴影的方法。需要注意的是，并不是所有的平台Unity都会使用这种技术。这是因为，屏幕空间的阴影映射需要显卡支持MRT，而有些移动平台不支持这种特性。

当使用了屏幕空间的阴影映射技术时，Unity首先会通过调用**LightMode**为**ShadowCaster**的Pass来得到可投射阴影的光源的阴影映射纹理以及摄像机的深度纹理。然后，根据光源的阴影映射纹理和摄像机的深度纹理来得到屏幕空间的阴影图。如果摄像机的深度图中记录的表面深度大于转换到阴影映射纹理中的深度值，就说明该表面虽然是可见的，但是却处于该光源的阴影中。通过这样的方式，阴影图就包含了屏幕空间中所有有阴影的区域。如果我们想要一个物体接收来自其他物体的阴影，只需要在Shader中对阴影图进行采样。由于阴影图是屏幕空间下的，因此，我们首先需要把表面坐标从模型空间变换到屏幕空间中，然后使用这个坐标对阴影图进行采样即可。

总结一下，一个物体接收来自其他物体的阴影，以及它向其他物体投射阴影是两个过程。

- 如果我们想要一个物体接收来自其他物体的阴影，就必须在**Shader**中对阴影映射纹理（包括屏幕空间的阴影图）进行采样，把采样结果和最后的光照结果相乘来产生阴影效果。
- 如果我们想要一个物体向其他物体投射阴影，就必须把该物体加入到光源的阴影映射纹理的计算中，从而让其他物体在对阴影映射纹理采样时可以得到该物体的相关信息。在**Unity**中，这个过程是通过为该物体执行**LightMode**为**ShadowCaster**的**Pass**来实现的。如果使用了屏幕空间的投影映射技术，**Unity**还会使用这个**Pass**产生一张摄像机的深度纹理。

在下面的章节中，我们会学习如何在**Unity**中实现上面两个过程。

### 9.4.2 不透明物体的阴影

我们首先进行如下的准备工作。

（1）在**Unity**中新建一个场景。在本书资源中，该场景名为**Scene\_9\_4\_2**。在**Unity 5.2**中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在**Window** → **Lighting** → **Skybox**中去掉场景中的天空盒子。

（2）新建一个材质。在本书资源中，该材质名为**ShadowMat**。我们把9.2节中的**Chapter9-ForwardRendering**赋给它。

（3）在场景中创建一个正方体、两个平面，并把第2步中的材质赋给正方体，但不改变两个平面的材质（默认情况下，它们会使用内置的**Standard**材质）。

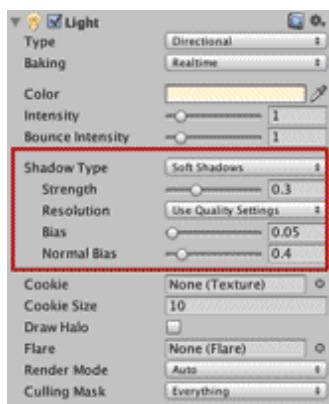
(4) 保存场景。

为了让场景中可以产生阴影，我们首先需要让平行光可以收集阴影信息。这需要在光源的**Light**组件中开启阴影，如图9.15所示。

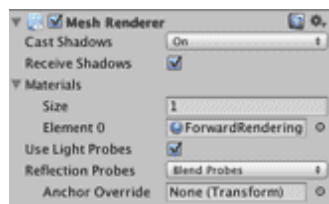
在本例中，我们选择了软阴影（**Soft Shadows**）。

## 1. 让物体投射阴影

在Unity中，我们可以选择是否让一个物体投射或接收阴影。这是通过设置**Mesh Renderer**组件中的**Cast Shadows**和**Receive Shadows**属性来实现的，如图9.16所示。



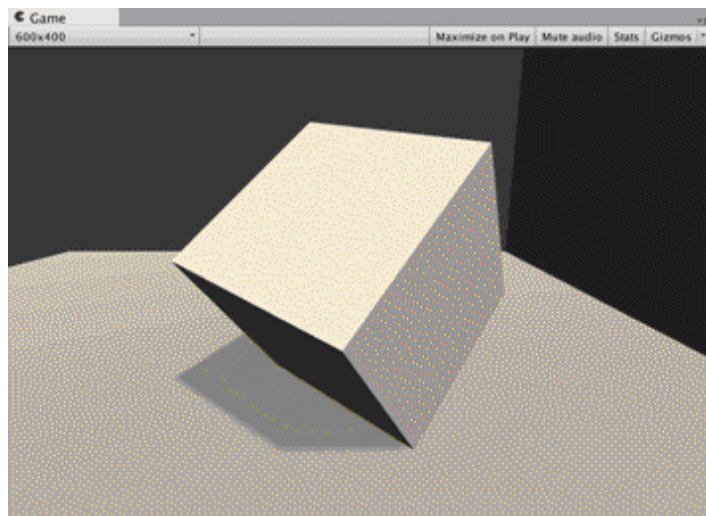
▲ 图9.15 开启光源的阴影效果



▲ 图9.16 Mesh Renderer组件的**Cast Shadows**和**Receive Shadows**属性可以控制该物体是否投射/接收阴影

**Cast Shadows**可以被设置为开启（On）或关闭（Off）。如果开启了**Cast Shadows**属性，那么Unity就会把该物体加入到光源的阴影映射纹理的计算中，从而让其他物体在对阴影映射纹理采样时可以得到该物体的相关信息。正如之前所说，这个过程是通过为该物体执行**LightMode**为**ShadowCaster**的Pass来实现的。**Receive Shadows**则可以选择是否让物体接收来自其他物体的阴影。如果没有开启**Receive Shadows**，那么当我们调用Unity的内置宏和变量计算阴影（在后面我们会看到如何实现）时，这些宏通过判断该物体没有开启接收阴影的功能，就不会在内部为我们计算阴影。

我们把正方体和两个平面的**Cast Shadows**和**Receive Shadows**都设为开启状态，可以得到图9.17中的结果。



▲ 图9.17 开启**Cast Shadows**和**Receive Shadows**，从而让正方体可以投射和接收阴影

从图9.17可以发现，尽管我们没有对正方体使用的Chapter9-**ForwardRendering**进行任何更改，但正方体仍然可以向下面的平面投射阴影。一些读者可能会有疑问：“之前不是说Unity要使用**LightMode**为

**ShadowCaster**的Pass来渲染阴影映射纹理和深度图吗？但是Chapter9-ForwardRendering中并没有这样一个Pass啊。”没错，我们在Chapter9-Forward Rendering的SubShader只定义了两个Pass——一个Base Pass，一个Additional Pass。那么为什么它还可以投射阴影呢？实际上，秘密就在于Chapter9- ForwardRendering中的**Fallback**语义：

Fallback "Specular"
---------------------

在Chapter9-ForwardRendering中，我们为它的Fallback指定了一个用于回调Unity Shader，即内置的Specular。虽然Specular本身也没有包含这样一个Pass，但是由于它的Fallback调用了VertexLit，它会继续回调，并最终回调到内置的VertexLit。我们可以在Unity内置的着色器里找到它：builtin-shaders-xxx->DefaultResourcesExtra->Normal-VertexLit.shader。打开它，我们就可以看到“传说中”的**LightMode**为**ShadowCaster**的Pass了：

```
// Pass to render object as a shadow caster
Pass {
    Name "ShadowCaster"
    Tags { "LightMode" = "ShadowCaster" }

    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag
    #pragma multi_compile_shadowcaster
    #include "UnityCG.cginc"

    struct v2f {
        V2F_SHADOW_CASTER;
    };

    v2f vert( appdata_base v )
    {
        v2f o;
        TRANSFER_SHADOW_CASTER_NORMALOFFSET(o)
        return o;
    }
}
```

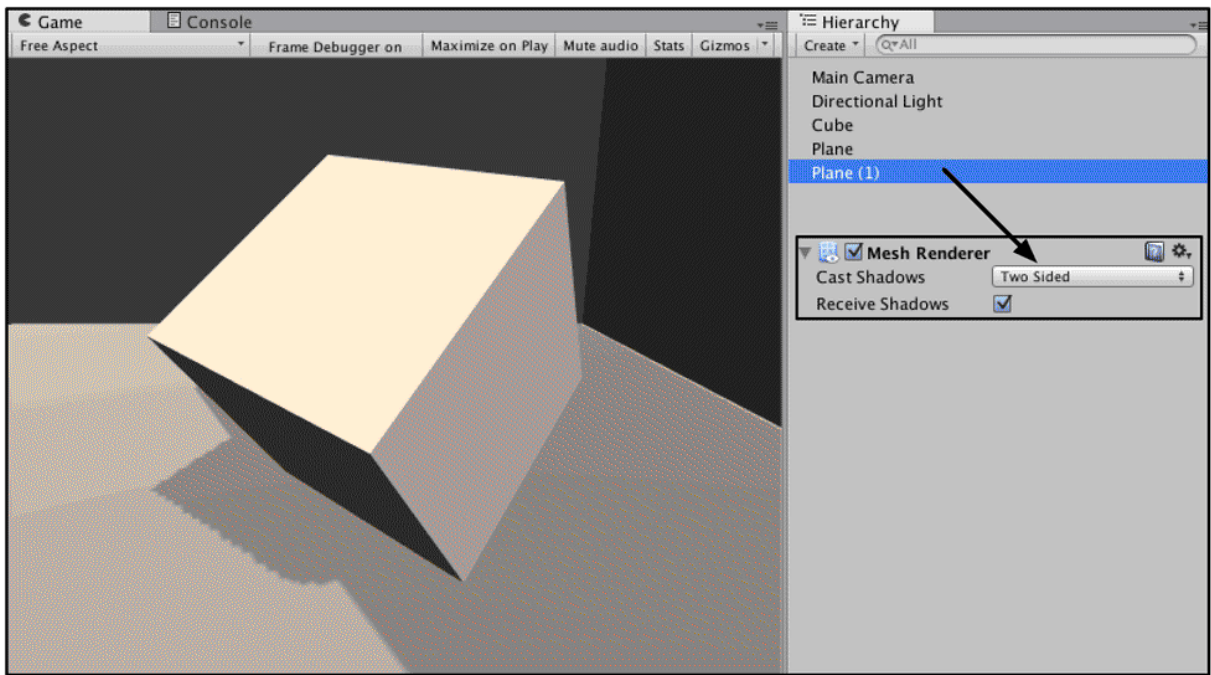
```
float4 frag( v2f i ) : SV_Target
{
    SHADOW_CASTER_FRAGMENT(i)
}
ENDCG
}
```

上面的代码非常短，尽管有一些宏和指令是我们之前没有遇到过的，但它们的用处实际上就是为了把深度信息写入渲染目标中。在Unity 5中，这个Pass的渲染目标可以是光源的阴影映射纹理，或是摄像机的深度纹理。

如果我们把Chapter9-ForwardRendering中的Fallback注释掉，就可以发现正方体不会再向平面投射阴影了。当然，我们可以不依赖Fallback，而自行在SubShader中定义自己的**LightMode**为**ShadowCaster**的Pass。这种自定义的Pass可以让我们更加灵活地控制阴影的产生。但由于这个Pass的功能通常是可以在多个Unity Shader间通用的，因此直接Fallback是一个更加方便的用法。在之前的章节中，我们有时也在Fallback中使用内置的Diffuse，虽然Diffuse本身也没有包含这样一个Pass，但是由于它的Fallback调用了VertexLit，因此Unity最终还是会找到一个**LightMode**为**ShadowCaster**的Pass，从而可以让物体产生阴影。在下面的9.4.2节中，我们将继续看到**LightMode**为**ShadowCaster**的Pass对产生正确的阴影的重要性。

图9.17中还有一个有意思的现象，就是右侧的平面并没有向最下面的平面投射阴影，尽管它的**Cast Shadows**已经被开启了。在默认情况下，我们在计算光源的阴影映射纹理时会剔除掉物体的背面。但对于内置的平面来说，它只有一个面，因此在本例中当计算阴影映射纹理

时，由于右侧的平面在光源空间下没有任何正面（frontface），因此就不会添加到阴影映射纹理中。我们可以将**Cast Shadows**设置为**Two Sided**来允许对物体的所有面都计算阴影信息。图9.18给出了当把右侧平面的**Cast Shadows**设置为**Two Sided**后的结果。



▲ 图9.18 把**Cast Shadows**设置为**Two Sided**可以让右侧平面的背光面也产生阴影

在本例中，最下面的平面之所以可以接收阴影是因为它使用了内置的**Standard Shader**，而这个内置的**Shader**进行了接收阴影的相关操作。但由于正方体使用的**Chapter9-ForwardRendering**并没有对阴影进行任何处理，因此它不会显示出右侧平面投射来的阴影。在下一节中，我们将学习如何让正方体也可以接收阴影。

## 2. 让物体接收阴影



为了让正方体可以接收阴影，我们首先新建一个Unity Shader，在本书资源中，它的名称为Chapter9-Shadow。我们把Chapter9-Shadow赋给正方体使用的材质ShadowMat。删除Chapter9-Shadow中的代码，把Chapter9-ForwardRendering的代码复制给它。当然，这样仍然不会有任何阴影出现在正方体上，因此我们需要对代码进行一些更改。

(1) 首先，我们在Base Pass中包含进一个新的内置文件：

```
#include "AutoLight.cginc"。
```

这是因为，我们下面计算阴影时所用的宏都是在这个文件中声明的。

(2) 我们在顶点着色器的输出结构体v2f中添加了一个内置宏**SHADOW\_COORDS**：

```
struct v2f {  
    float4 pos : SV_POSITION;  
    float3 worldNormal : TEXCOORD0;  
    float3 worldPos : TEXCOORD1;  
    SHADOW_COORDS(2)  
};
```

这个宏的作用很简单，就是声明一个用于对阴影纹理采样的坐标。需要注意的是，这个宏的参数需要是下一个可用的插值寄存器的索引值，在上面的例子中就是2。

(3) 然后，我们在顶点着色器返回之前添加另一个内置宏**TRANSFER\_SHADOW**：

```
v2f vert(a2v v) {  
    v2f o;  
    ...
```

```

    // Pass shadow coordinates to pixel shader
    TRANSFER_SHADOW(o);

    return o;
}

```

这个宏用于在顶点着色器中计算上一步中声明的阴影纹理坐标。

(4) 接着，我们在片元着色器中计算阴影值，这同样使用了一个内置宏**SHADOW\_ATTENUATION**：

```

// Use shadow coordinates to sample shadow map
fixed shadow = SHADOW_ATTENUATION(i);

```

**SHADOW\_COORDS**、**TRANSFER\_SHADOW**和**SHADOW\_ATTENUATION**是计算阴影时的“三剑客”。这些内置宏帮助我们在必要时计算光源的阴影。我们可以在AutoLight.cginc中找到它们的声明：

```

    // -----
    // Shadow helpers
    // -----

    // ---- Screen space shadows
    #if defined (SHADOWS_SCREEN)
        UNITY_DECLARE_SHADOWMAP(_ShadowMapTexture);
        #define SHADOW_COORDS(idx1) unityShadowCoord4 _ShadowCoord :
        TEXCOORD##idx1;
        #if defined(UNITY_NO_SCREENSPACE_SHADOWS)
            #define TRANSFER_SHADOW(a) a._ShadowCoord = mul(
            unity_World2Shadow[0],
            mul( _Object2World, v.vertex ) );
            inline fixed unitySampleShadow (unityShadowCoord4
            shadowCoord)
            {
                ...
            }
        #else // UNITY_NO_SCREENSPACE_SHADOWS
            #define TRANSFER_SHADOW(a) a._ShadowCoord =
            ComputeScreenPos(a.pos);
            inline fixed unitySampleShadow (unityShadowCoord4
            shadowCoord)

```

```

        {
            fixed shadow = tex2Dproj( _ShadowMapTexture,
UNITY_PROJ_COORD(shadowCoord) ).r;
            return shadow;
        }
    #endif
    #define SHADOW_ATTENUATION(a) unitySampleShadow(a._ShadowCoord)
#endif

// ---- Spot light shadows
#if defined (SHADOWS_DEPTH) && defined (SPOT)
    ...
#endif

// ---- Point light shadows
#if defined (SHADOWS_CUBE)
    ...
#endif

// ---- Shadows off
#if !defined (SHADOWS_SCREEN) && !defined (SHADOWS_DEPTH) &&
!defined (SHADOWS_CUBE)
    #define SHADOW_COORDS(idx1)
    #define TRANSFER_SHADOW(a)
    #define SHADOW_ATTENUATION(a) 1.0
#endif

```

上面的代码看起来很多、很复杂，实际上只是Unity为了处理不同光源类型、不同平台而定义了多个版本的宏。在前向渲染中，宏**SHADOWCOORDS**实际上就是声明了一个名为\_ShadowCoord的阴影纹理坐标变量。而**TRANSFER\_SHADOW**的实现会根据平台不同而有所差异。如果当前平台可以使用屏幕空间的阴影映射技术（通过判断是否定义了**UNITY\_NO\_SCREENSPACE SHADOWS**来得到），**TRANSFER\_SHADOW**会调用内置的ComputeScreenPos函数来计算\_ShadowCoord；如果该平台不支持屏幕空间的阴影映射技术，就会使用传统的阴影映射技术，**TRANSFER\_SHADOW**会把顶点坐标从模型空间变换到光源空间后存储到\_ShadowCoord中。然后，

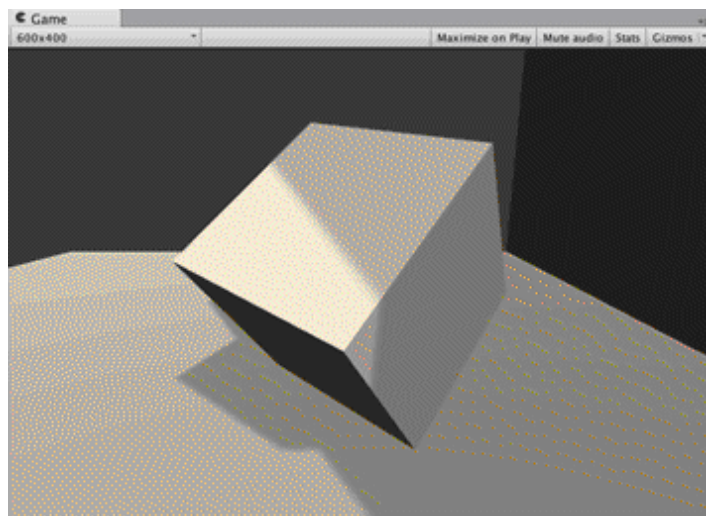
SHADOW\_ATTENUATION负责使用\_ShadowCoord对相关的纹理进行采样，得到阴影信息。

注意到，上面内置代码的最后定义了关闭阴影时的处理代码。可以看出，当关闭了阴影后，**SHADOW\_COORDS**和**TRANSFER\_SHADOW**实际没有任何作用，而**SHADOW\_ATTENUATION**会直接等同于数值1。

**需要读者注意的是**，由于这些宏中会使用上下文变量来进行相关计算，例如TRANSFER\_SHADOW会使用v.vertex或a.pos来计算坐标，因此为了能够让这些宏正确工作，我们需要保证自定义的变量名和这些宏中使用的变量名相匹配。我们需要保证：a2v结构体的顶点坐标变量名必须是**vertex**，顶点着色器的输入结构体a2v必须命名为**v**，且v2f中的顶点位置变量必须命名为**pos**。

(5) 在完成了上面的所有操作后，我们只需要把阴影值shadow和漫反射以及高光反射颜色相乘即可。

保存文件，返回Unity我们可以发现，现在正方体也可以接收来自右侧平面的阴影了，如图9.19所示。



▲ 图9.19 正方体可以接收来自右侧平面的阴影

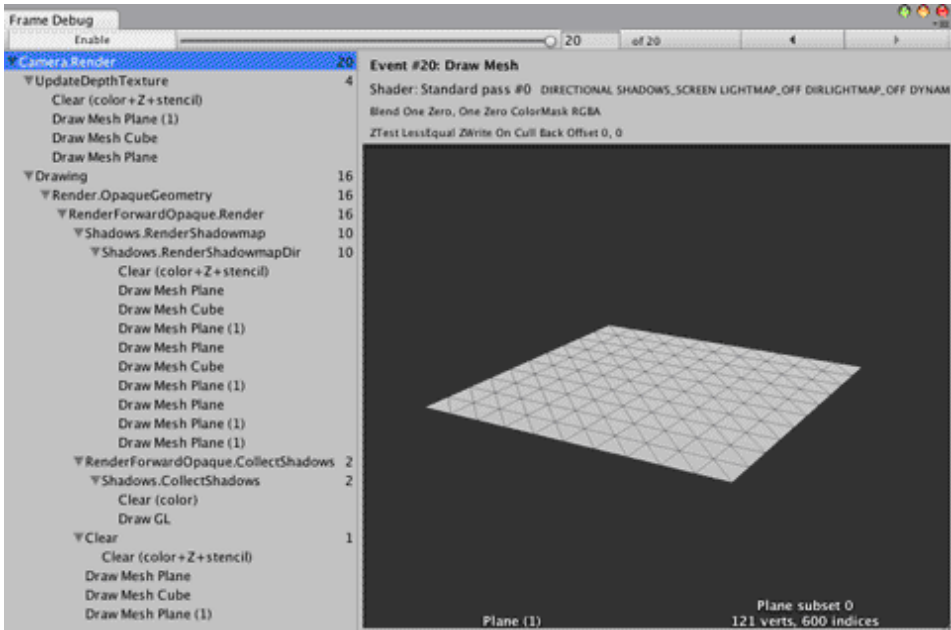
需要注意的是，在上面的代码里我们只更改了**Base Pass**中的代码，使其可以得到阴影效果，而没有对**Additional Pass**进行任何更改。大体上，**Additional Pass**的阴影处理和**Base Pass**是一样的。我们将在9.4.4节看到如何处理这些阴影。本节实现的代码仅是为了解释如何让物体接收阴影，但不可以直接应用到项目中。我们会在9.5节中给出包含了完整的光照处理的Unity Shader。

### 9.4.3 使用帧调试器查看阴影绘制过程

尽管我们在上面描述了阴影的产生过程，但如果有直观的方式看到阴影一步步的绘制过程那就太好了！幸运的是，Unity 5添加了一个新的工具——帧调试器。我们曾在9.2.2节中利用它查看过Pass的绘制过程，在本节我们会通过它来查看阴影的绘制过程。

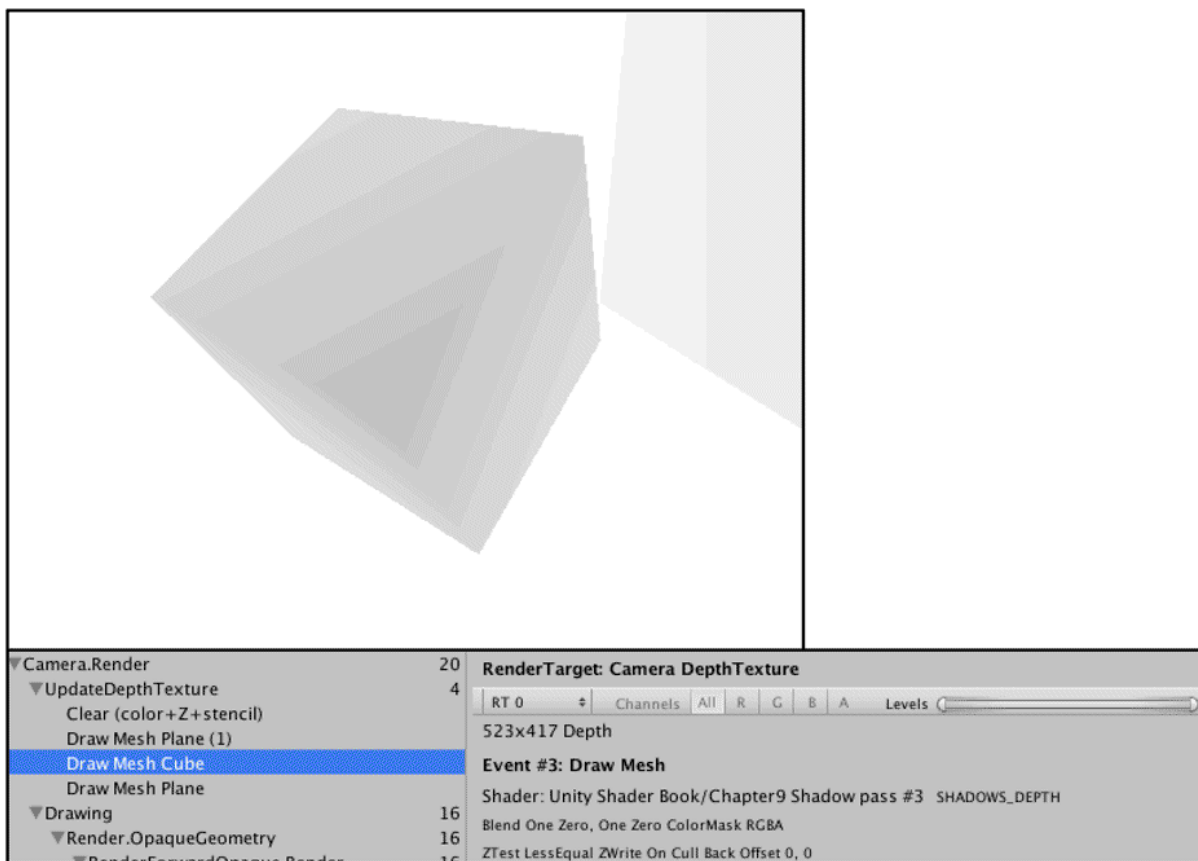
首先，我们需要在Window -> Frame Debugger中打开帧调试器。图9.20给出了Scene\_9\_4\_2在帧调试器中的分析结果。

从图9.20中可以看出，绘制该场景共需要花费20个渲染事件。这些渲染事件可以分为4个部分：UpdateDepthTexture，即更新摄像机的深度纹理；RenderShadowmap，即渲染得到平行光的阴影映射纹理；CollectShadows，即根据深度纹理和阴影映射纹理得到屏幕空间的阴影图；最后绘制渲染结果。



▲ 图9.20 使用帧调试器查看阴影绘制过程

我们首先来看第一个部分：更新摄像机的深度纹理，这是前4个渲染事件的工作。我们可以单击这些事件查看它们的绘制结果。图9.21给出了正方体对深度纹理的更新结果。



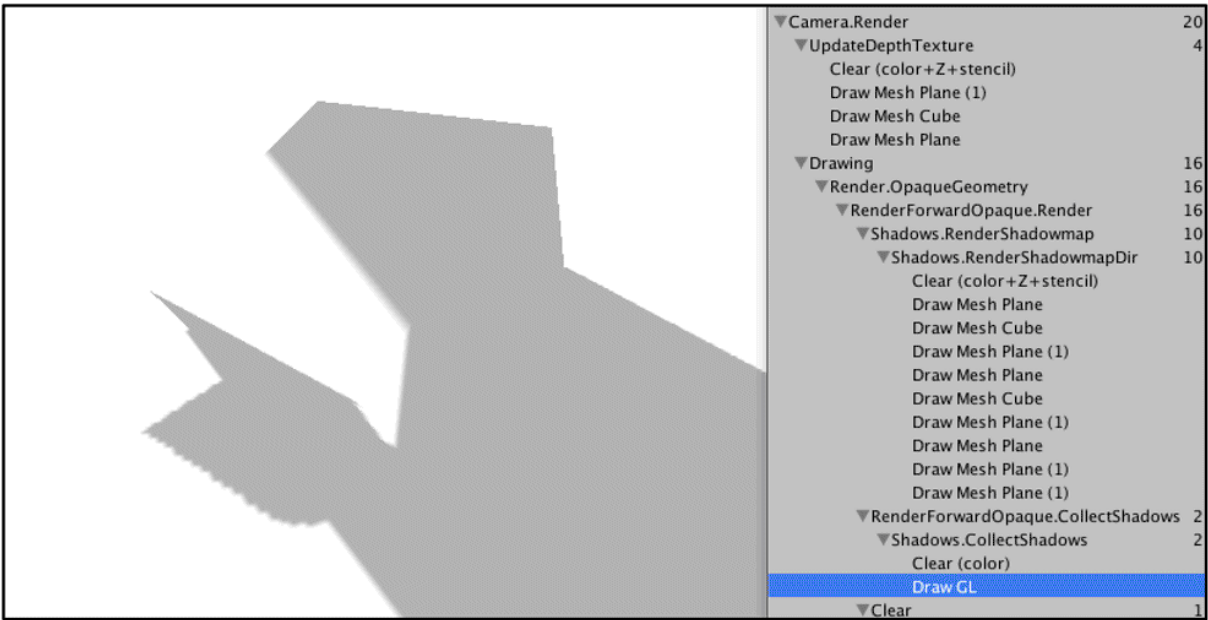
▲ 图9.21 正方体对深度纹理的更新结果

从帧调试器右侧的面板我们可以了解这一渲染事件的详细信息。在图9.21中，我们可以发现，Unity调用了**Shader: Unity Shader Book/Chapter9 Shadow pass #3**来更新深度纹理，即Chapter9-Shadow中的第三个Pass。尽管Chapter9-Shadow中只定义了两个Pass，但正如我们之前所说，Unity会在它的Fallback中找到第三个Pass，即**LightMode**为**ShadowCaster**的Pass来更新摄像机的深度纹理。同样，在第二个部分，即渲染得到平行光的阴影映射纹理的过程中，Unity也是调用了这个Pass来得到光源的阴影映射纹理。

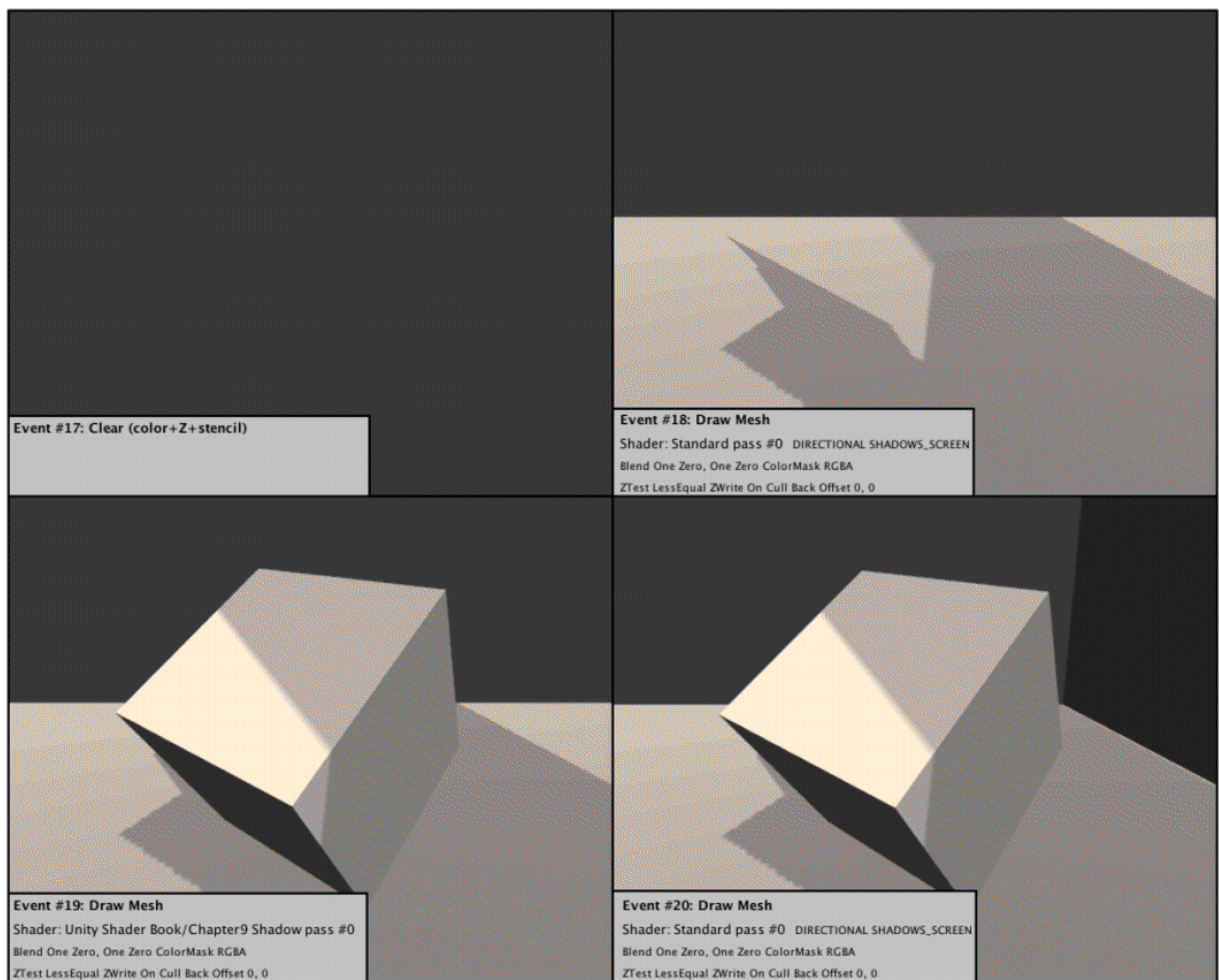


在第三个部分中，Unity会根据之前两步的结果得到屏幕空间的阴影图，如图9.22所示。

这张图已经包含了最终屏幕上所有有阴影区域的阴影。在最后一个部分中，如果物体所使用的Shader包含了对这张阴影图的采样就会得到阴影效果。图9.23给出了这个部分Unity是如何一步步绘制出有阴影的画面效果的。



▲图9.22 屏幕空间的阴影图



▲ 图9.23 Unity绘制屏幕阴影的过程

#### 9.4.4 统一管理光照衰减和阴影

在9.2节和9.3节中，我们已经讲过如何在Unity Shader的前向渲染路径中计算光照衰减——在Base Pass中，平行光的衰减因子总是等于1，而在Additional Pass中，我们需要判断该Pass处理的光源类型，再使用内置变量和宏计算衰减因子。实际上，光照衰减和阴影对物体最终的渲染结果的影响本质上是相同的——我们都是把光照衰减因子和阴影值及光照结果相乘得到最终的渲染结果。那么，是不是可以有一个方法可以同时计算两个信息呢？好消息是，Unity在Shader里提供了这样

的功能，这主要是通过内置的**UNITY\_LIGHT\_ATTENUATION**宏来实现的。

为此，我们做如下准备工作。

(1) 复制9.4.2节中同样的场景，在本书资源中该场景名为**Scene\_9\_4\_4**。

(2) 新建一个材质。在本书资源中，该材质名为**AttenuationAndShadowUseBuildInFunctionsMat**。

(3) 新建一个Unity Shader。在本书资源中，该Shader名为**Chapter9-AttenuationAndShadowUse BuildInFunctions**。把新的Shader赋给第2步中创建的材质。

(4) 把第2步中的材质赋给一个正方体。

(5) 保存场景。

打开**Chapter9-AttenuationAndShadowUseBuildInFunctions**，把**Chapter9-Shadow**中的代码粘贴进去。尽管**Chapter9-Shadow**中的代码可以让我们得到正确的阴影，但在实践中我们通常会使用Unity的内置宏和函数来计算衰减和阴影，从而隐藏一些实现细节。关键代码如下。

(1) 首先包含进需要的头文件。

```
// Need these files to get built-in macros
#include "Lighting.cginc"
#include "AutoLight.cginc"
```

(2) 在v2f结构体中使用内置宏**SHADOW\_COORDS**声明阴影坐标:

```
struct v2f {
    float4 pos : SV_POSITION;
    float3 worldNormal : TEXCOORD0;
    float3 worldPos : TEXCOORD1;
    SHADOW_COORDS(2)
};
```

(3) 在顶点着色器中使用内置宏**TRANSFER\_SHADOW**计算并向片元着色器传递阴影坐标:

```
v2f vert(a2v v) {
    v2f o;
    ...
    TRANSFER_SHADOW(o);

    return o;
}
```

(4) 和9.4.2节中的方式不同, 这次我们在片元着色器中使用内置宏**UNITYLIGHT\_ATTENUATION**来计算光照衰减和阴影:

```
fixed4 frag(v2f i) : SV_Target {
    ...

    // UNITY_LIGHT_ATTENUATION not only compute attenuation, but
    also shadow infos
    UNITY_LIGHT_ATTENUATION(atten, i, i.worldPos);

    return fixed4(ambient + (diffuse + specular) * atten, 1.0);
}
```

**UNITYLIGHT\_ATTENUATION**是Unity内置的用于计算光照衰减和阴影的宏, 我们可以在内置的AutoLight.cginc里找到它的相关声明。它接受3个参数, 它会将光照衰减和阴影值相乘后的结果存储到第一个参数中。注意到, 我们并没有在代码中声明第一个参数**atten**, 这是因

为**UNITY\_LIGHT\_ATTENUATION**会帮我们声明这个变量。它的第二个参数是结构体**v2f**，这个参数会传递给9.4.2节中使用的**SHADOW\_ATTENUATION**，用来计算阴影值。而第三个参数是世界空间的坐标，正如我们在9.3节中看到的一样，这个参数会用于计算光源空间下的坐标，再对光照衰减纹理采样来得到光照衰减。我们强烈建议读者查阅AutoLight.cginc中**UNITY\_LIGHT\_ATTENUATION**的声明，读者可以发现，Unity针对不同光源类型、是否启用cookie等不同情况声明了多个版本的**UNITY\_LIGHT\_ATTENUATION**。这些不同版本的声明是保证我们可以通过这样一个简单的代码来得到正确结果的关键。

(5) 由于使用了**UNITY\_LIGHT\_ATTENUATION**，我们的Base Pass和Additional Pass的代码得以统一——我们不需要在Base Pass里单独处理阴影，也不需要Additional Pass中判断光源类型来处理光照衰减，一切都只需要通过**UNITY\_LIGHT\_ATTENUATION**来完成即可。这正是Unity内置文件的魅力所在。如果我们希望可以在Additional Pass中添加阴影效果，就需要使用**#pragma multi\_compile\_fwdadd\_fullshadows**编译指令来代替Additional Pass中的**#pragma multi\_compile\_fwdadd**指令。这样一来，Unity也会为这些额外的逐像素光源计算阴影，并传递给Shader。

### 9.4.5 透明度物体的阴影

我们从一开始就强调，想要在Unity里让物体能够向其他物体投射阴影，一定要在它使用的Unity Shader中提供一个**LightMode**为**ShadowCaster**的Pass。在前面的例子中，我们使用内置的**VertexLit**中提供的**ShadowCaster**来投射阴影。**VertexLit**中的**ShadowCaster**实现很

简单，它会正常渲染整个物体，然后把深度结果输出到一张深度图或阴影映射纹理中。读者可以在内置文件中找到相关的文件。

对于大多数不透明物体来说，把**Fallback**设为**VertexLit**就可以得到正确的阴影。但对于透明物体来说，我们就需要小心处理它的阴影。透明物体的实现通常会使用透明度测试或透明度混合，我们需要小心设置这些物体的**Fallback**。

透明度测试的处理比较简单，但如果我们仍然直接使用**VertexLit**、**Diffuse**、**Specular**等作为回调，往往无法得到正确的阴影。这是因为透明度测试需要在片元着色器中舍弃某些片元，而**VertexLit**中的阴影投射纹理并没有进行这样的操作。我们在本书资源的**Scene\_9\_4\_5\_a**中提供了这样一个测试场景。我们使用了之前学习的透明度测试 + 阴影的方法来渲染一个正方体，它使用的材质和Unity Shader分别是**AlphaTestWithShadowMat**和**Chapter9-AlphaTestWithShadow**。Chapter9-AlphaTestWithShadow使用了和8.3节透明度测试中几乎完全相同的代码，只是添加了关于阴影的计算。

(1) 首先包含进需要的头文件：

```
#include "Lighting.cginc"
#include "AutoLight.cginc"
```

(2) 在v2f中使用内置宏**SHADOW\_COORDS**声明阴影纹理坐标：

```
struct v2f {
    float4 pos : SV_POSITION;
    float3 worldNormal : TEXCOORD0;
    float3 worldPos : TEXCOORD1;
    float2 uv : TEXCOORD2;
```

```
SHADOW_COORDS(3)
};
```

注意到，由于我们已经占用了 3 个插值寄存器（使用 `TEXCOORD0`、`TEXCOORD1`和`TEXCOORD2`修饰的变量），因此 **SHADOW\_COORDS**中传入的参数是3，这意味着，阴影纹理坐标将占用第四个插值寄存器`TEXCOORD3`。

（3）然后，在顶点着色器中使用内置宏**TRANSFER\_SHADOW**计算阴影纹理坐标后传递给片元着色器：

```
v2f vert(a2v v) {
    v2f o;
    ...
    // Pass shadow coordinates to pixel shade
    TRANSFER_SHADOW(o);

    return o;
}
```

（4）在片元着色器中，使用内置宏**UNITY\_LIGHT\_ATTENUATION**计算阴影和光照衰减：

```
fixed4 frag(v2f i) : SV_Target {
    ...
    // UNITY_LIGHT_ATTENUATION not only compute attenuation, but
    also shadow infos
    UNITY_LIGHT_ATTENUATION(atten, i, i.worldPos);

    return fixed4(ambient + diffuse * atten, 1.0);
}
```

（5）这次，我们更改它的Fallback，使用**VertexLit**作为它的回调Shader：

```
Fallback "VertexLit"
```

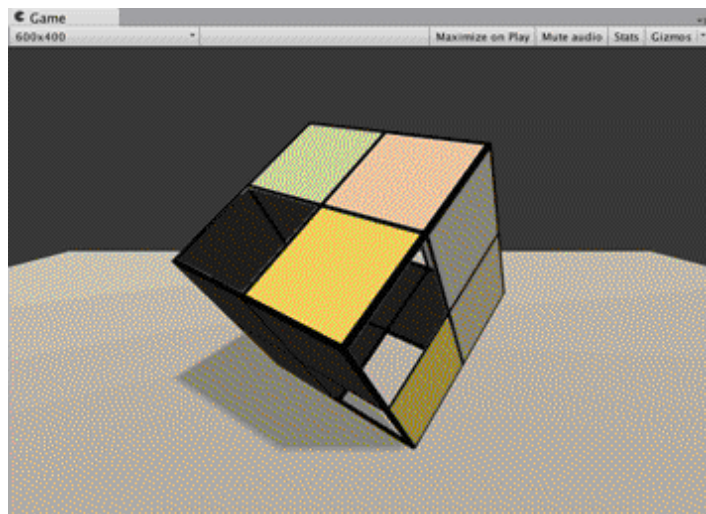


我们仍然使用transparent\_texture.psd纹理，把它赋给新的材质后，就可以得到类似图9.24中的效果。

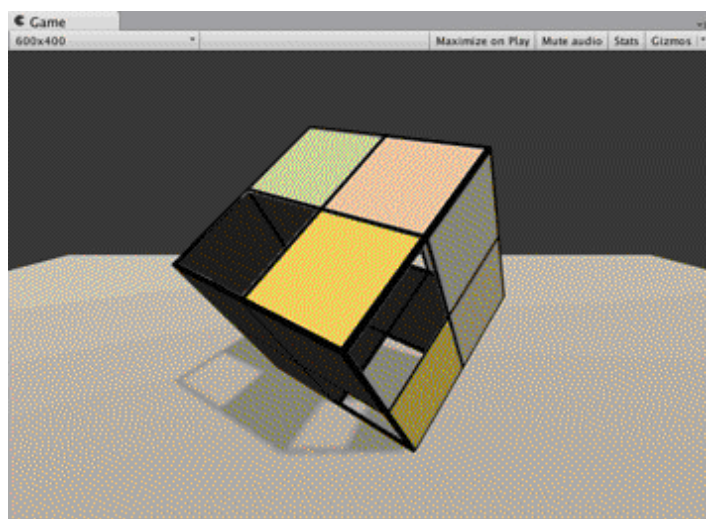
细心的读者可以发现，镂空区域出现了不正常的阴影，看起来就像这个正方体是一个普通的正方体一样。而这并不是我们想要得到的，我们希望有些光应该是可以通过这些镂空区域透过来的，这些区域不应该有阴影。出现这样的情况是因为，我们使用的是内置的**VertexLit**中提供的**ShadowCaster**来投射阴影，而这个**Pass**中并没有进行任何透明度测试的计算，因此，它会把整个物体的深度信息渲染到深度图和阴影映射纹理中。因此，如果我们想要得到经过透明度测试后的阴影效果，就需要提供一个有透明度测试功能的**ShadowCaster Pass**。当然，我们可以自行编写一个这样的**Pass**，但这里我们仍然选择使用内置的Unity Shader来减少代码量。

为了让使用透明度测试的物体得到正确的阴影效果，我们只需要在Unity Shader中更改一行代码，即把**Fallback**设置为**Transparent/Cutout/VertexLit**，正如我们在8.3节中实现的一样。读者可以在内置文件中找到该Unity Shader的代码，它的**ShadowCaster Pass**也计算了透明度测试，因此会把裁剪后的物体深度信息写入深度图和阴影映射纹理中。**但需要注意的是**，由于**Transparent/Cutout/VertexLit**中计算透明度测试时，使用了名为**\_Cutoff**的属性来进行透明度测试，因此，这要求我们的Shader中也必须提供名为**\_Cutoff**的属性。否则，同样无法得到正确的阴影结果。

在更改了**Fallback**后，我们可以得到图9.25中的效果。



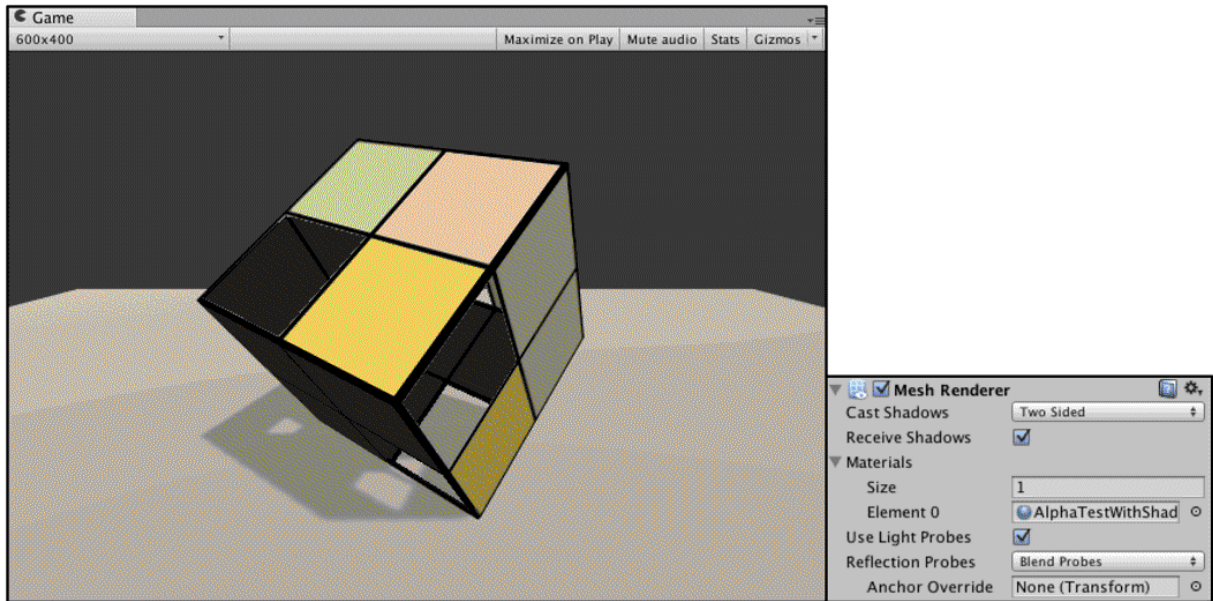
▲ 图9.24 可以投射阴影的使用透明度测试的物体



▲ 图9.25 正确设置了Fallback的使用透明度测试的物体

但是，这样的结果仍然有一些问题，例如出现了一些不应该透光的部分。出现这种情况的原因是，默认情况下把物体渲染到深度图和阴影映射纹理中仅考虑物体的正面。但对于本例的正方体来说，由于一些面完全背对光源，因此这些面的深度信息没有加入到阴影映射纹理的计算中。为了得到正确的结果，我们可以将正方体的Mesh Renderer组件中的Cast Shadows属性设置为**Two Sided**，强制Unity在计

算阴影映射纹理时计算所有面的深度信息。图9.26给出了正确设置后的渲染结果。

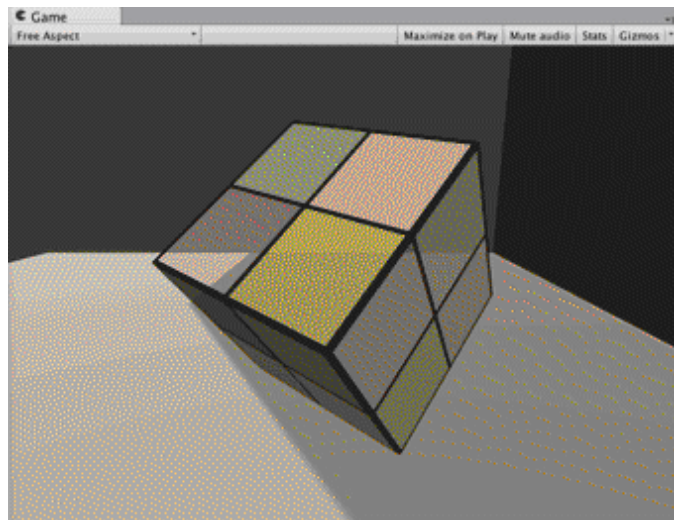


▲ 图9.26 正确设置了Cast Shadow属性的使用透明度测试的物体

与透明度测试的物体相比，想要为使用透明度混合的物体添加阴影是一件比较复杂的事情。事实上，所有内置的透明度混合的Unity Shader，如Transparent/VertexLit等，都没有包含阴影投射的Pass。这意味着，这些半透明物体不会参与深度图和阴影映射纹理的计算，也就是说，它们不会向其他物体投射阴影，同样它们也不会接收来自其他物体的阴影。我们在本书资源的Scene\_9\_4\_5\_b中提供了这样一个测试场景。我们使用了之前学习的透明度混合 + 阴影的方法来渲染一个正方体，它使用的材质和 Unity Shader 分别是 AlphaBlendWithShadowMat 和 Chapter9-AlphaBlendWithShadow。Chapter9-AlphaBlendWithShadow 使用了和8.4节透明度混合中几乎完全相同的代码，只是添加了关于阴

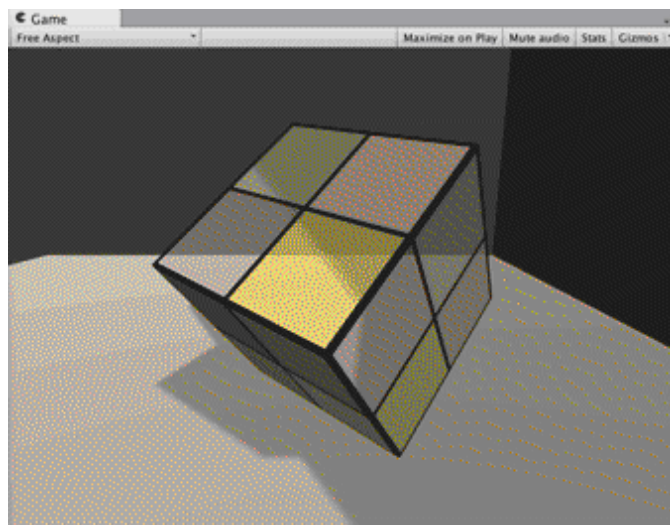
影的计算，并且它的Fallback是内置的Transparent/VertexLit。图9.27显示了渲染结果。

Unity会这样处理半透明物体是有它的原因的。由于透明度混合需要关闭深度写入，由此带来的问题也影响了阴影的生成。总体来说，要想为这些半透明物体产生正确的阴影，需要在每个光源空间下仍然严格按照从后往前的顺序进行渲染，这会让阴影处理变得非常复杂，而且也会影响性能。因此，在Unity中，所有内置的半透明Shader是不会产生任何阴影效果的。当然，我们可以使用一些dirty trick来强制为半透明物体生成阴影，这可以通过把它们的Fallback设置为VertexLit、Diffuse这些不透明物体使用的Unity Shader，这样Unity就会在它的Fallback找到一个阴影投射的Pass。然后，我们可以通过物体的Mesh Renderer组件上的Cast Shadows和Receive Shadows选项来控制是否需要向其他物体投射或接收阴影。图9.28显示了把Fallback设为VertexLit并开启阴影投射和接收阴影后的半透明物体的渲染效果。



▲ 图9.27 把使用了透明度混合的Unity Shader的Fallback设置为内置的Transparent/VertexLit。半透明物体不会向下方的平面投射阴影，也不会接收来自右侧平面的阴影，它看起来就像是完

全透明一样



▲ 图9.28 把Fallback设为VertexLit来强制为半透明物体生成阴影

可以看出，此时右侧平面的阴影投射到了半透明的立方体上，但它不会再穿透立方体把阴影投射到下方的平面上，这其实是不正确的。同时，立方体也可以把自身的阴影投射到下面的平面上。

## 9.5 本书使用的标准Unity Shader

到了实现诺言的时候了！我们在之前的实现中一直强调，这些代码仅仅是为了阐述Unity中的各种光照实现原理，由于缺少一些光照计算，因此不可以直接使用到项目中。截止到本节，我们已经学习了Unity中所有的基础光照计算，如多光源、阴影和光照衰减等。现在是时候把它们整合到一起来实现一个标准光照着色器了！我们在本书资源的Assets/ Shaders/Common文件夹下提供了两个这样标准的Unity Shader——BumpedDiffuse和BumpedSpecular。这两个Unity Shader都包含了对法线纹理、多光源、光照衰减和阴影的相关处理，唯一不同的



是，BumpedDiffuse使用了Phong光照模型，而BumpedSpecular使用了Blinn-Phong光照模型。读者可以打开这两个文件，此时可以发现里面的代码都是我们学习过的。我们使用这两个Unity Shader创建了多个材质（在Assets/Material/Objects和Assets/Material/Walls文件夹下），这些材质将被用于后面章节的场景搭建中。读者可以参考这两个Unity Shader来实现透明版本的Unity Shader。

## 第10章 高级纹理

我们在第7章学习了关于基础纹理的内容，这些纹理包括法线纹理、渐变纹理和遮罩纹理等。这些纹理尽管用处不同，但它们都属于低维（一维或二维）纹理。在本章中，我们将学习一些更复杂的纹理。在10.1节中，我们会学习如何使用立方体纹理（Cubemap）实现环境映射。然后，我们会在10.2节介绍一类特殊的纹理——渲染纹理（Render Texture），我们会发现渲染纹理是多么的强大。最后，10.3节将介绍程序纹理（Procedure Texture）。

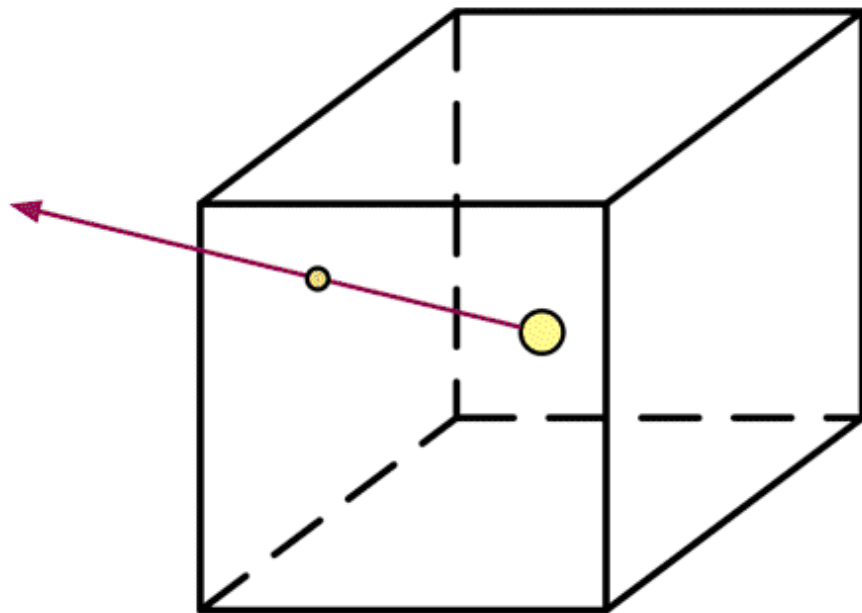
### 10.1 立方体纹理

在图形学中，**立方体纹理（Cubemap）**是**环境映射（Environment Mapping）**的一种实现方法。环境映射可以模拟物体周围的环境，而使用了环境映射的物体可以看起来像镀了层金属一样反射出周围的环境。

和之前见到的纹理不同，立方体纹理一共包含了6张图像，这些图像对应了一个立方体的6个面，立方体纹理的名称也由此而来。立方体的每个面表示沿着世界空间下的轴向（上、下、左、右、前、后）观察所得的图像。那么，我们如何对这样一种纹理进行采样呢？和之前使用二维纹理坐标不同，对立方体纹理采样我们需要提供一个三维的纹理坐标，这个三维纹理坐标表示了我们在世界空间下的一个3D方向。这个方向矢量从立方体的中心出发，当它向外部延伸时就会和立



方体的6个纹理之一发生相交，而采样得到的结果就是由该交点计算而来的。图10.1给出了使用方向矢量对立方体纹理采样的过程。



▲图10.1 对立方体纹理的采样

使用立方体纹理的好处在于，它的实现简单快速，而且得到的效果也比较好。但它也有一些缺点，例如当场景中引入了新的物体、光源，或者物体发生移动时，我们就需要重新生成立方体纹理。除此之外，立方体纹理也仅可以反射环境，但不能反射使用了该立方体纹理的物体本身。这是因为，立方体纹理不能模拟多次反射的结果，例如两个金属球互相反射的情况（事实上，Unity 5引入的全局光照系统允许实现这样的自反射效果，详见第18章）。由于这样的原因，想要得到令人信服的渲染结果，我们应该尽量对凸面体而不要对凹面体使用立方体纹理（因为凹面体会反射自身）。

立方体纹理在实时渲染中有很多应用，最常见的是用于天空盒子（Skybox）以及环境映射。

### 10.1.1 天空盒子

**天空盒子（Skybox）**是游戏中用于模拟背景的一种方法。天空盒子这个名字包含了两个信息：它是用来模拟天空的（尽管现在我们仍可以用它模拟室内等背景），它是一个盒子。当我们在场景中使用了天空盒子时，整个场景就被包围在一个立方体内。这个立方体的每个面使用的技术就是立方体纹理映射技术。

在Unity中，想要使用天空盒子非常简单。我们只需要创建一个Skybox材质，再把它赋给该场景的相关设置即可。

我们首先来看如何创建一个Skybox材质。

- （1）新建一个材质，在本书资源中该材质名为SkyboxMat。
- （2）在SkyboxMat的Unity Shader下拉菜单中选择Unity自带的Skybox/6 Sided，该材质需要6张纹理。
- （3）使用本书资源中的Assets/Textures/Chapter10/Cubemaps文件夹下的6张纹理对第2步中的材质赋值，注意这6张纹理的正确位置（如posz纹理对应了Front [+Z] 属性）。为了让天空盒子正常渲染，我们需要把这6张纹理的**Wrap Mode**设置为**Clamp**，以防止在接缝处出现不匹配的现象。

上述步骤得到的材质如图10.2所示。



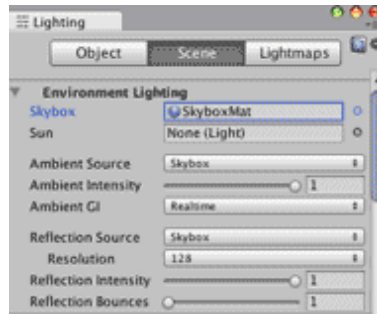
▲ 图10.2 天空盒子材质

上面的材质中，除了6张纹理属性外还有3个属性：**Tint Color**，用于控制该材质的整体颜色；**Exposure**，用于调整天空盒子的亮度；**Rotation**，用于调整天空盒子沿+y轴方向的旋转角度。

下面，我们来看一下如何为场景添加Skybox。

- (1) 新建一个场景，在本书资源中该场景名为Scene\_10\_1\_1。
- (2) 在Window → Lighting菜单中，把SkyboxMat赋给Skybox选项，如图10.3所示。

为了让摄像机正常显示天空盒子，我们还需要保证渲染场景的摄像机的Camera组件中的Clear Flags被设置为**Skybox**。这样，我们得到的场景如图10.4所示。



▲图10.3 为场景使用自定义的天空盒子



▲图10.4 使用了天空盒子的场景

需要说明的是，在Window → Lighting → Skybox中设置的天空盒子会应用于该场景中的所有摄像机。如果我们希望某些摄像机可以使用不同的天空盒子，可以通过向该摄像机添加**Skybox**组件来覆盖掉之前的设置。也就是说，我们可以在摄像机上单击Component → Rendering → Skybox来完成对场景默认天空盒子的覆盖。

在Unity中，天空盒子是在所有不透明物体之后渲染的，而其背后使用的网格是一个立方体或一个细分后的球体。

### 10.1.2 创建用于环境映射的立方体纹理

除了天空盒子，立方体纹理最常见的用处是用于环境映射。通过这种方法，我们可以模拟出金属质感的材质。

在Unity 5中，创建用于环境映射的立方体纹理的方法有三种：第一种方法是直接由一些特殊布局的纹理创建；第二种方法是手动创建一个Cubemap资源，再把6张图赋给它；第三种方法是由脚本生成。

如果使用第一种方法，我们需要提供一张具有特殊布局的纹理，例如类似立方体展开图的交叉布局、全景布局等。然后，我们只需要把该纹理的**Texture Type**设置为**Cubemap**即可，Unity会为我们做好剩下的事情。在基于物理的渲染中，我们通常会使用一张HDR图像来生成高质量的Cubemap（详见第18章）。读者可在官方文档

（<http://docs.unity3d.com/Manual/class-Cubemap.html>）中找到更多的资料。

第二种方法是Unity 5之前的版本中使用的方法。我们首先需要在项目资源中创建一个Cubemap，然后把6张纹理拖曳到它的面板中。在Unity 5中，官方推荐使用第一种方法创建立方体纹理，这是因为第一种方法可以对纹理数据进行压缩，而且可以支持边缘修正、光滑反射（glossy reflection）和HDR等功能。

前面两种方法都需要我们提前准备好立方体纹理的图像，它们得到的立方体纹理往往是被场景中的物体所共用的。但在理想情况下，

我们希望根据物体在场景中位置的不同，生成它们各自不同的立方体纹理。这时，我们就可以在Unity中使用脚本来创建。这是通过利用Unity提供的**Camera.RenderToCubemap**函数来实现的。

**Camera.RenderToCubemap**函数可以把从任意位置观察到的场景图像存储到6张图像中，从而创建出该位置上对应的立方体纹理。

在Unity的脚本手册

(<http://docs.unity3d.com/ScriptReference/Camera.RenderToCubemap.html>) 中给出了如何使用**Camera.RenderToCubemap**函数来创建立方体纹理的代码。读者也可以在本书资源的

**Assets/Editor/Chapter10/RenderCubemapWizard.cs**中找到相关代码。其中关键代码如下：

```
void OnWizardCreate () {
    // create temporary camera for rendering
    GameObject go = new GameObject( "CubemapCamera");
    go.AddComponent<Camera>();
    // place it on the object
    go.transform.position = renderFromPosition.position;
    // render into cubemap
    go.GetComponent<Camera>().RenderToCubemap(cubemap);

    // destroy temporary camera
    DestroyImmediate( go );
}
```

在上面的代码中，我们在**renderFromPosition**（由用户指定）位置处动态创建一个摄像机，并调用**Camera.RenderToCubemap**函数把从当前位置观察到的图像渲染到用户指定的立方体纹理**cubemap**中，完成后再销毁临时摄像机。由于该代码需要添加菜单栏条目，因此我们需要把它放在**Editor**文件夹下才能正确执行。

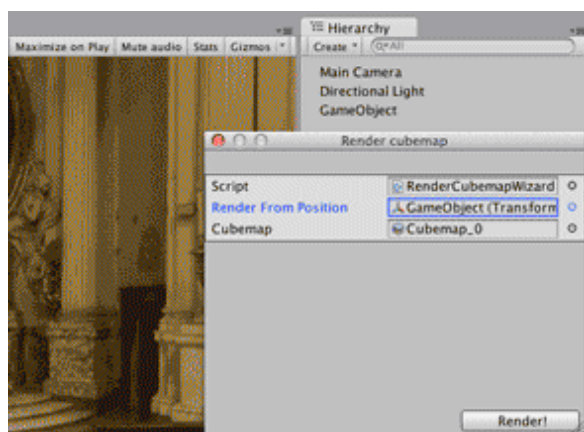
当准备好上述代码后，要创建一个**Cubemap**非常简单。

(1) 我们使用和10.1.1节中相同的场景，并创建一个空的GameObject对象。我们会使用该GameObject的位置信息来渲染立方体纹理。

(2) 新建一个用于存储的立方体纹理（在Project视图下单击右键，选择Create → Legacy → Cubemap来创建）。在本书资源中，该立方体纹理名为Cubemap\_0。为了让脚本可以顺利将图像渲染到该立方体纹理中，我们需要在它的面板中勾选Readable选项。

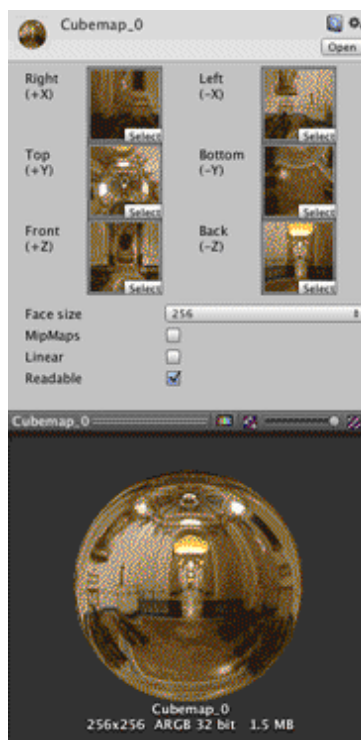
(3) 从Unity菜单栏选择GameObject -> Render into Cubemap，打开我们在脚本中实现的用于渲染立方体纹理的窗口，并把第1步中创建的GameObject和第2步中创建的Cubemap\_0分别拖曳到窗口中的**Render From Position**和**Cubemap**选项，如图10.5所示。

(4) 单击窗口中的**Render!**按钮，就可以把从该位置观察到的世界空间下的6张图像渲染到Cubemap\_0中，如图10.6所示。



▲ 图10.5 使用脚本创建立方体纹理





▲ 图10.6 使用脚本渲染立方体纹理

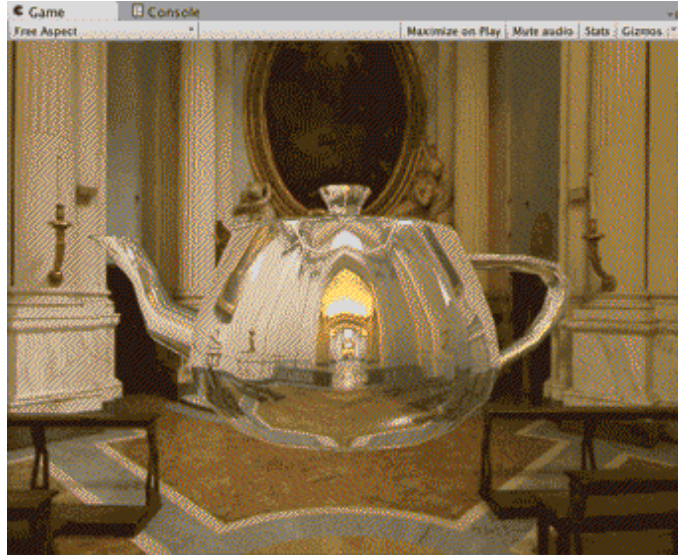
需要注意的是，我们需要为Cubemap设置大小，即图10.6中的**Face size**选项。**Face size**值越大，渲染出来的立方体纹理分辨率越大，效果可能更好，但需要占用的内存也越大，这可以由面板最下方显示的内存大小得到。

准备好了需要的立方体纹理后，我们就可以对物体使用环境映射技术。而环境映射最常见的应用就是反射和折射。

### 10.1.3 反射

使用了反射效果的物体通常看起来就像镀了层金属。想要模拟反射效果很简单，我们只需要通过入射光线的方向和表面法线方向来计

算反射方向，再利用反射方向对立方体纹理采样即可。在学习完本节  
后，我们可以得到类似图10.7中的效果。



▲ 图10.7 使用了反射效果的Teapot模型

为此，我们需要做如下准备工作。

(1) 新建一个场景，在本书资源中，该场景名为Scene\_10\_1\_3。我们替换掉Unity 5中场景默认的天空盒子，而把10.1.1节中创建的天空盒子材质拖曳到Window → Lighting → Skybox选项中（当然，我们也可以为摄像机添加Skybox组件来覆盖默认的天空盒子）。

(2) 向场景中拖曳一个Teapot模型，并调整它的位置和10.1.2节中创建Cubemap\_0时使用的空GameObject的位置相同。

(3) 新建一个材质，在本书资源中，该材质名为ReflectionMat，把材质赋给第2步中创建的Teapot模型。

(4) 新建一个Unity Shader，在本书资源中，该Shader名为Chapter10-Reflection。把Chapter10-Reflection赋给第3步中创建的材质。

反射的实现非常简单。打开Chapter10-Reflection，删除原有的代码，进行如下关键修改。

(1) 首先，我们声明了3个新的属性：

```
Properties {  
    _Color ("Color Tint", Color) = (1, 1, 1, 1)  
    _ReflectColor ("Reflection Color", Color) = (1, 1, 1, 1)  
    _ReflectAmount ("Reflect Amount", Range(0, 1)) = 1  
    _Cubemap ("Reflection Cubemap", Cube) = "_Skybox" {}  
}
```

其中，\_ReflectColor用于控制反射颜色，\_ReflectAmount用于控制这个材质的反射程度，而\_Cubemap就是用于模拟反射的环境映射纹理。

(2) 我们在顶点着色器中计算了该顶点处的反射方向，这是通过使用CG的**reflect**函数来实现的：

```
v2f vert(a2v v) {  
    v2f o;  
  
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);  
  
    o.worldNormal = UnityObjectToWorldNormal(v.normal);  
  
    o.worldPos = mul(_Object2World, v.vertex).xyz;  
  
    o.worldViewDir = UnityWorldSpaceViewDir(o.worldPos);  
  
    // Compute the reflect dir in world space  
    o.worldRefl = reflect(-o.worldViewDir, o.worldNormal);  
  
    TRANSFER_SHADOW(o);  
}
```

```
    return o;  
}
```

物体反射到摄像机中的光线方向，可以由光路可逆的原则来反向求得。也就是说，我们可以计算视角方向关于顶点法线的反射方向来求得入射光线的方向。

(3) 在片元着色器中，利用反射方向来对立方体纹理采样：

```
fixed4 frag(v2f i) : SV_Target {  
    fixed3 worldNormal = normalize(i.worldNormal);  
    fixed3 worldLightDir =  
normalize(UnityWorldSpaceLightDir(i.worldPos));  
    fixed3 worldViewDir = normalize(i.worldViewDir);  
  
    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;  
  
    fixed3 diffuse = _LightColor0.rgb * _Color.rgb * max(0,  
dot(worldNormal, worldLightDir));  
  
    // Use the reflect dir in world space to access the cubemap  
    fixed3 reflection = texCUBE(_Cubemap, i.worldRefl).rgb *  
_ReflectColor.rgb;  
  
    UNITY_LIGHT_ATTENUATION(atten, i, i.worldPos);  
  
    // Mix the diffuse color with the reflected color  
    fixed3 color = ambient + lerp(diffuse, reflection,  
_ReflectAmount) * atten;  
  
    return fixed4(color, 1.0);  
}
```

对立方体纹理的采样需要使用CG的**texCUBE**函数。注意到，在上面的计算中，我们在采样时并没有对*i.worldRefl*进行归一化操作。这是因为，用于采样的参数仅仅是作为方向变量传递给**texCUBE**函数的，因此我们没有必要进行一次归一化的操作。然后，我们使用**\_ReflectAmount**来混合漫反射颜色和反射颜色，并和环境光照相加后返回。

在上面的计算中，我们选择在顶点着色器中计算反射方向。当然，我们也可以选择在片元着色器中计算，这样得到的效果更加细腻。但是，对于绝大多数人来说这种差别往往是可以忽略不计的，因此出于性能方面的考虑，我们选择在顶点着色器中计算反射方向。

保存后返回场景，在材质面板中把Cubemap\_0拖曳到**Reflection Cubemap**属性中，并调整其他参数，即可得到类似图10.7中的效果。

### 10.1.4 折射

在这一节中，我们将学习如何在Unity Shader中模拟另一个环境映射的常见应用——折射。

折射的物理原理比反射复杂一些。我们在初中物理就已经接触过折射的定义：当光线从一种介质（例如空气）斜射入另一种介质（例如玻璃）时，传播方向一般会发生改变。当给定入射角时，我们可以使用**斯涅尔定律（Snell's Law）**来计算反射角。当光从介质1沿着和表面法线夹角为 $\theta_1$ 的方向斜射入介质2时，我们可以使用如下公式计算折射光线与法线的夹角 $\theta_2$ ：

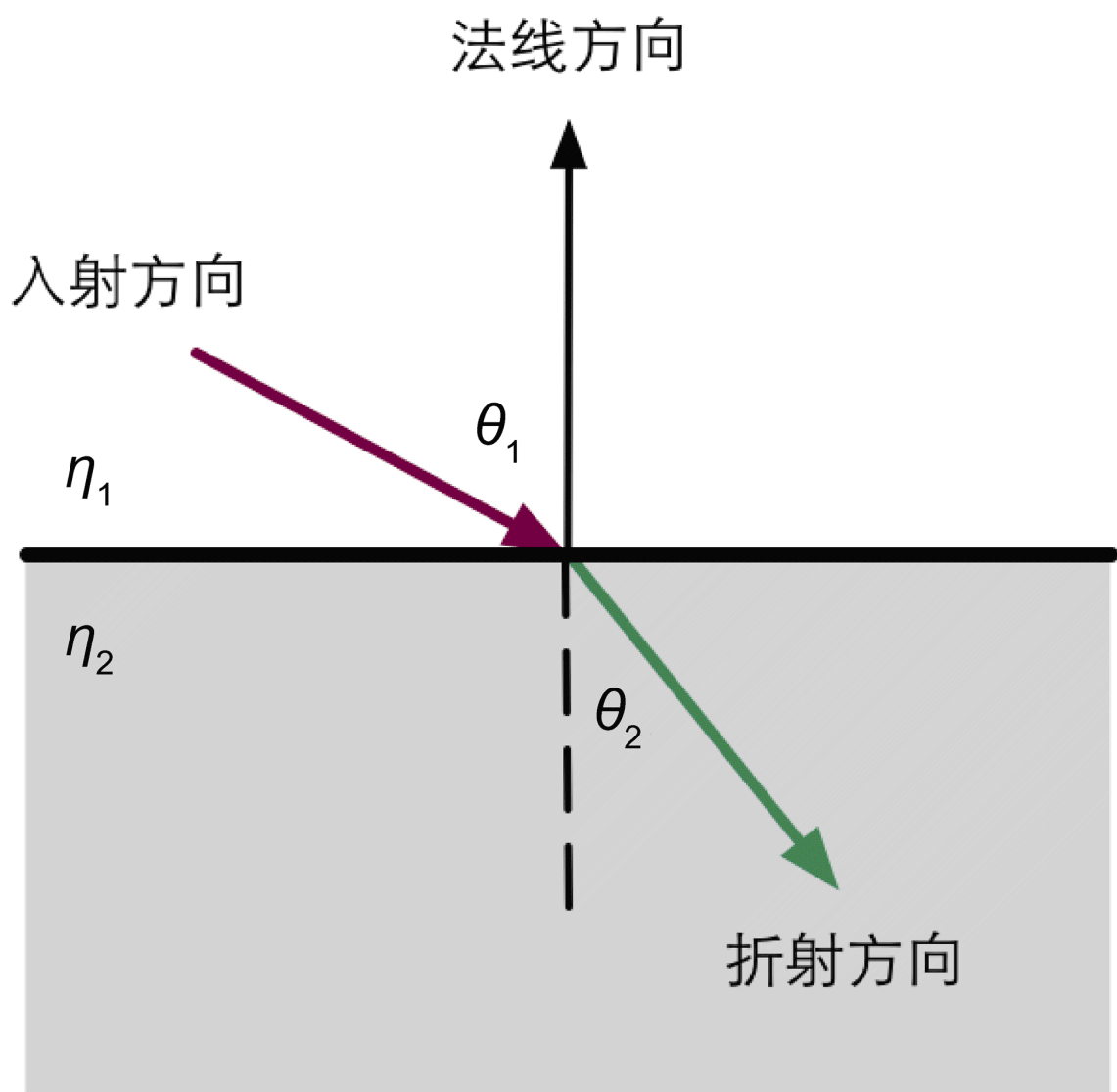
$$\eta_1 \sin \theta_1 = \eta_2 \sin \theta_2$$

其中， $\eta_1$ 和 $\eta_2$ 分别是两个介质的**折射率（index of refraction）**。折射率是一项重要的物理常数，例如真空的折射率是1，而玻璃的折射率一般是1.5。图10.8给出了这些变量之间的关系。

通常来说，当得到折射方向后我们会直接使用它来对立方体纹理进行采样，但这是不符合物理规律的。对一个透明物体来说，一种

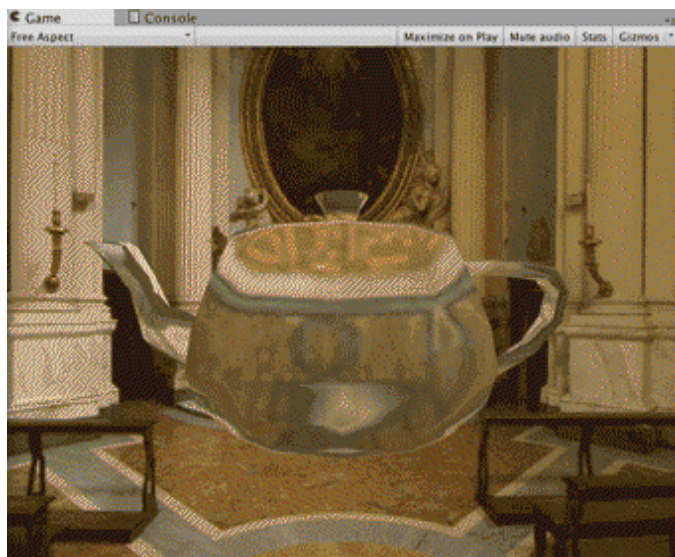
更准确的模拟方法需要计算两次折射——一次是当光线进入它的内部时，而另一次则是从它内部射出时。但是，想要在实时渲染中模拟出第二次折射方向是比较复杂的，而且仅仅模拟一次得到的效果从视觉上看起来“也挺像那么回事的”。正如我们之前提到的——图形学第一准则“如果它看起来是对的，那么它就是对”。因此，在实时渲染中我们通常仅模拟第一次折射。

在学习完本节后，我们可以得到类似图10.9中的效果。



▲图10.8 斯涅尔定律





▲ 图10.9 使用了折射效果的Teapot模型

为此，我们需要做如下准备工作。

(1) 新建一个场景，在本书资源中，该场景名为**Scene\_10\_1\_4**。我们替换掉Unity 5中场景默认的天空盒子，而把10.1.1节中创建的天空盒子材质拖曳到**Window → Lighting → Skybox**选项中（当然，我们也可以为摄像机添加**Skybox**组件来覆盖默认的天空盒子）。

(2) 向场景中拖曳一个**Teapot**模型，并调整它的位置。

(3) 新建一个材质，在本书资源中，该材质名为**RefractionMat**，把材质赋给第2步中创建的**Teapot**模型。

(4) 新建一个Unity Shader，在本书资源中，该Shader名为**Chapter10-Refraction**。把**Chapter10- Refraction**赋给第3步中创建的材质。

折射效果的实现略微复杂一些。打开Chapter10-Refraction，删除原有的代码，进行如下关键修改。

(1) 首先，我们声明了4个新属性：

```
Properties {
    _Color ("Color Tint", Color) = (1, 1, 1, 1)
    _RefractColor ("Refraction Color", Color) = (1, 1, 1, 1)
    _RefractAmount ("Refraction Amount", Range(0, 1)) = 1
    _RefractRatio ("Refraction Ratio", Range(0.1, 1)) = 0.5
    _Cubemap ("Refraction Cubemap", Cube) = "_Skybox" {}
}
```

其中，\_RefractColor、\_RefractAmount和\_Cubemap与10.1.3节中控制反射时使用的属性类似。除此之外，我们还使用了一个属性 \_RefractRatio，我们需要使用该属性得到不同介质的透射比，以此来计算折射方向。

(2) 在顶点着色器中，计算折射方向：

```
v2f vert(a2v v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.worldNormal = UnityObjectToWorldNormal(v.normal);

    o.worldPos = mul(_Object2World, v.vertex).xyz;

    o.worldViewDir = UnityWorldSpaceViewDir(o.worldPos);

    // Compute the refract dir in world space
    o.worldRefr = refract(-normalize(o.worldViewDir),
normalize(o.worldNormal), _RefractRatio);

    TRANSFER_SHADOW(o);

    return o;
}
```

我们使用了CG的**refract**函数来计算折射方向。它的第一个参数即为入射光线的方向，它必须是归一化后的矢量；第二个参数是表面法线，法线方向同样需要是归一化后的；第三个参数是入射光线所在介质的折射率和折射光线所在介质的折射率之间的比值，例如如果光是从空气射到玻璃表面，那么这个参数应该是空气的折射率和玻璃的折射率之间的比值，即1/1.5。它的返回值就是计算而得的折射方向，它的模则等于入射光线的模。

(3) 然后，我们在片元着色器中使用折射方向对立方体纹理进行采样：

```
fixed4 frag(v2f i) : SV_Target {
    fixed3 worldNormal = normalize(i.worldNormal);
    fixed3 worldLightDir =
normalize(UnityWorldSpaceLightDir(i.worldPos));
    fixed3 worldViewDir = normalize(i.worldViewDir);

    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;

    fixed3 diffuse = _LightColor0.rgb * _Color.rgb * max(0,
dot(worldNormal, worldLightDir));

    // Use the refract dir in world space to access the cubemap
    fixed3 refraction = texCUBE(_Cubemap, i.worldRefr).rgb *
_RefractColor.rgb;

    UNITY_LIGHT_ATTENUATION(atten, i, i.worldPos);

    // Mix the diffuse color with the refract color
    fixed3 color = ambient + lerp(diffuse, refraction,
_RefractAmount) * atten;

    return fixed4(color, 1.0);
}
```

同样，我们也没有对*i.worldRefr*进行归一化操作，因为对立方体纹理的采样只需要提供方向即可。最后，我们使用\_RefractAmount来混合漫反射颜色和折射颜色，并和环境光照相加后返回。

保存后返回场景，在材质面板中把Cubemap\_0拖曳到**Reflection Cubemap**属性中，并调整其他参数，即可得到类似图10.9中的效果。

### 10.1.5 菲涅耳反射

在实时渲染中，我们经常会使用**菲涅耳反射（Fresnel reflection）**来根据视角方向控制反射程度。通俗地讲，菲涅耳反射描述了一种光学现象，即当光线照射到物体表面上时，一部分发生反射，一部分进入物体内部，发生折射或散射。被反射的光和入射光之间存在一定的比率关系，这个比率关系可以通过菲涅耳等式进行计算。一个经常使用的例子是，当你站在湖边，直接低头看脚边的水面时，你会发现水几乎是透明的，你可以直接看到水底的小鱼和石子；但是，当你抬头看远处的水面时，会发现几乎看不到水下情景，而只能看到水面反射的环境。这就是所谓的菲涅耳效果。事实上，不仅仅是水、玻璃这样的反光物体具有菲涅耳效果，几乎任何物体都或多或少包含了菲涅耳效果，这是基于物理的渲染中非常重要的一项高光反射计算因子

（详见第18章）。读者可以在John Hable的一篇非常有名的文章***Everything Has Fresnel***（<http://filmicgames.com/archives/557>）中看到现实生活中各种物体的菲涅耳效果。

那么，我们如何计算菲涅耳反射呢？这就需要使用菲涅耳等式。真实世界的菲涅耳等式是非常复杂的，但在实时渲染中，我们通常会使用一些近似公式来计算。其中一个著名的近似公式就是**Schlick菲涅耳近似等式**：

$$F_{Schlick}(\mathbf{v}, \mathbf{n}) = F_0 + (1 - F_0)(1 - \mathbf{v} \cdot \mathbf{n})^5$$

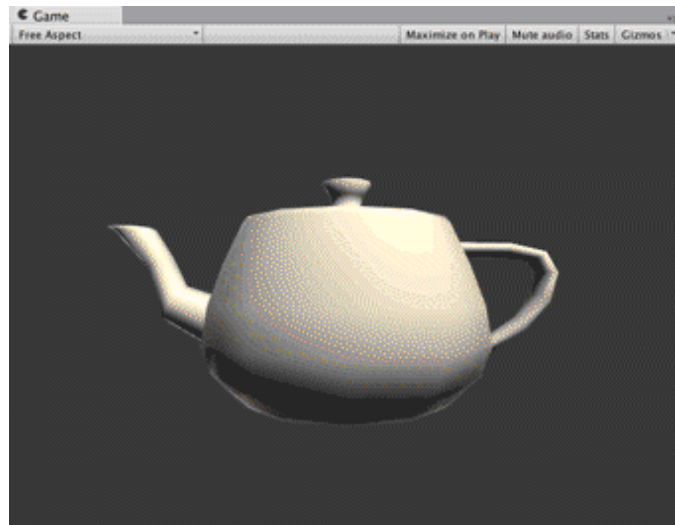
其中， $F_0$ 是一个反射系数，用于控制菲涅耳反射的强度， $\mathbf{v}$ 是视角方向， $\mathbf{n}$ 是表面法线。另一个应用比较广泛的等式是**Empirical**菲涅耳近似等式：

$$F_{Empirical}(\mathbf{v}, \mathbf{n}) = \max(0, \min(1, \text{bias} + \text{scale} \times (1 - \mathbf{v} \cdot \mathbf{n})^{\text{power}}))$$

其中， $\text{bias}$ 、 $\text{scale}$ 和 $\text{power}$ 是控制项。

使用上面的菲涅耳近似等式，我们可以在边界处模拟反射光强和折射光强/漫反射光强之间的变化。在许多车漆、水面等材质的渲染中，我们会经常使用菲涅耳反射来模拟更加真实的反射效果。

在本节中，我们将使用**Schlick**菲涅耳近似等式来模拟菲涅耳反射。在本节最后，我们可以得到类似图10.10中的效果。注意图中在模型边界处的反射现象。



▲ 图10.10 使用了菲涅耳反射的Teapot模型

为此，我们需要做如下准备工作。

(1) 新建一个场景，在本书资源中，该场景名为Scene\_10\_1\_5。我们替换掉Unity 5中场景默认的天空盒子，而把10.1.1节中创建的天空盒子材质拖曳到Window → Lighting → Skybox选项中（当然，我们也可以为摄像机添加Skybox组件来覆盖默认的天空盒子）。

(2) 向场景中拖曳一个Teapot模型，并调整它的位置。

(3) 新建一个材质，在本书资源中，该材质名为FresnelMat，把材质赋给第2步中创建的Teapot模型。

(4) 新建一个Unity Shader，在本书资源中，该Shader名为Chapter10-Fresnel。把Chapter10- Fresnel赋给第3步中创建的材质。

打开Chapter10-Fresnel，删除原有的代码，进行如下关键修改。

(1) 首先，我们在Properties语义块中声明了用于调整菲涅耳反射的属性以及反射使用的Cubemap：

```
Properties {  
    _Color ("Color Tint", Color) = (1, 1, 1, 1)  
    _FresnelScale ("Fresnel Scale", Range(0, 1)) = 0.5  
    _Cubemap ("Reflection Cubemap", Cube) = "_Skybox" {}  
}
```

(2) 在顶点着色器中计算世界空间下的法线方向、视角方向和反射方向：

```
v2f vert(a2v v) {  
    v2f o;  
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);  
  
    o.worldNormal = mul(v.normal, (float3x3)_World2Object);  
  
    o.worldPos = mul(_Object2World, v.vertex).xyz;
```

```

    o.worldViewDir = UnityWorldSpaceViewDir(o.worldPos);

    o.worldRefl = reflect(-o.worldViewDir, o.worldNormal);

    TRANSFER_SHADOW(o);

    return o;
}

```

(3) 在片元着色器中计算菲涅耳反射，并使用结果值混合漫反射光照和反射光照：

```

fixed4 frag(v2f i) : SV_Target {
    fixed3 worldNormal = normalize(i.worldNormal);
    fixed3 worldLightDir =
normalize(UnityWorldSpaceLightDir(i.worldPos));
    fixed3 worldViewDir = normalize(i.worldViewDir);

    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;

    UNITY_LIGHT_ATTENUATION(atten, i, i.worldPos);

    fixed3 reflection = texCUBE(_Cubemap, i.worldRefl).rgb;

    fixed fresnel = _FresnelScale + (1 - _FresnelScale) * pow(1 -
dot(worldViewDir, worldNormal), 5);

    fixed3 diffuse = _LightColor0.rgb * _Color.rgb * max(0,
dot(worldNormal, worldLightDir));

    fixed3 color = ambient + lerp(diffuse, reflection,
saturate(fresnel)) * atten;

    return fixed4(color, 1.0);
}

```

在上面的代码中，我们使用Schlick菲涅耳近似等式来计算fresnel变量，并使用它来混合漫反射光照和反射光照。一些实现也会直接把fresnel和反射光照相乘后叠加到漫反射光照上，模拟边缘光照的效果。



保存后返回场景，在材质面板中把Cubemap\_0拖曳到**Cubemap**属性中，并调整其他参数，即可得到类似图10.10中的效果。当我们把\_FresnelScale调节到1时，物体将完全反射Cubemap中的图像；当\_FresnelScale为0时，则是一个具有边缘光照效果的漫反射物体。我们还会在15.2节中使用菲涅耳反射来混合反射和折射光照，以此来模拟一个简单的水面效果。

## 10.2 渲染纹理

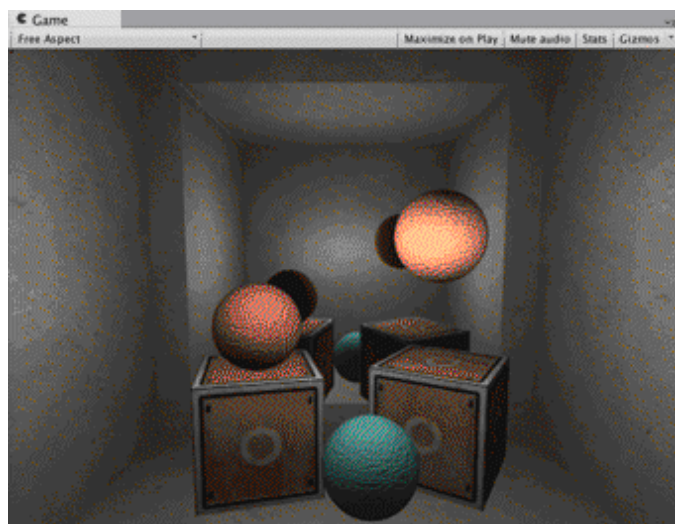
在之前的学习中，一个摄像机的渲染结果会输出到颜色缓冲中，并显示到我们的屏幕上。现代的GPU允许我们把整个三维场景渲染到一个中间缓冲中，即**渲染目标纹理（Render Target Texture, RTT）**，而不是传统的帧缓冲或后备缓冲（back buffer）。与之相关的是**多重渲染目标（Multiple Render Target, MRT）**，这种技术指的是GPU允许我们把场景同时渲染到多个渲染目标纹理中，而不再需要为每个渲染目标纹理单独渲染完整的场景。延迟渲染就是使用多重渲染目标的一个应用。

Unity为渲染目标纹理定义了一种专门的纹理类型——**渲染纹理（Render Texture）**。在Unity中使用渲染纹理通常有两种方式：一种方式是在Project目录下创建一个渲染纹理，然后把某个摄像机的渲染目标设置成该渲染纹理，这样一来该摄像机的渲染结果就会实时更新到渲染纹理中，而不会显示在屏幕上。使用这种方法，我们还可以选择渲染纹理的分辨率、滤波模式等纹理属性。另一种方式是在屏幕后处理时使用GrabPass命令或OnRenderImage函数来获取当前屏幕图像，Unity会把这个屏幕图像放到一张和屏幕分辨率等同的渲染纹理中，下

面我们可以在自定义的Pass中把它们当成普通的纹理来处理，从而实现各种屏幕特效。我们将依次学习这两种方法在Unity中的实现（OnRenderImage函数会在第12章中讲到）。

### 10.2.1 镜子效果

在本节中，我们将学习如何使用渲染纹理来模拟镜子效果。学习完本节后，我们可以得到类似图10.11中的效果。



▲ 图10.11 镜子效果

为此，我们需要做如下准备工作。

（1）新建一个场景。在本书资源中，该场景名为Scene\_10\_2\_1。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window → Lighting → Skybox中去掉场景中的天空盒子。

（2）新建一个材质。在本书资源中，该材质名为MirrorMat。

(3) 新建一个Unity Shader。在本书资源中，该Shader名为Chapter10-Mirror。把新的Shader赋给第2步中创建的材质。

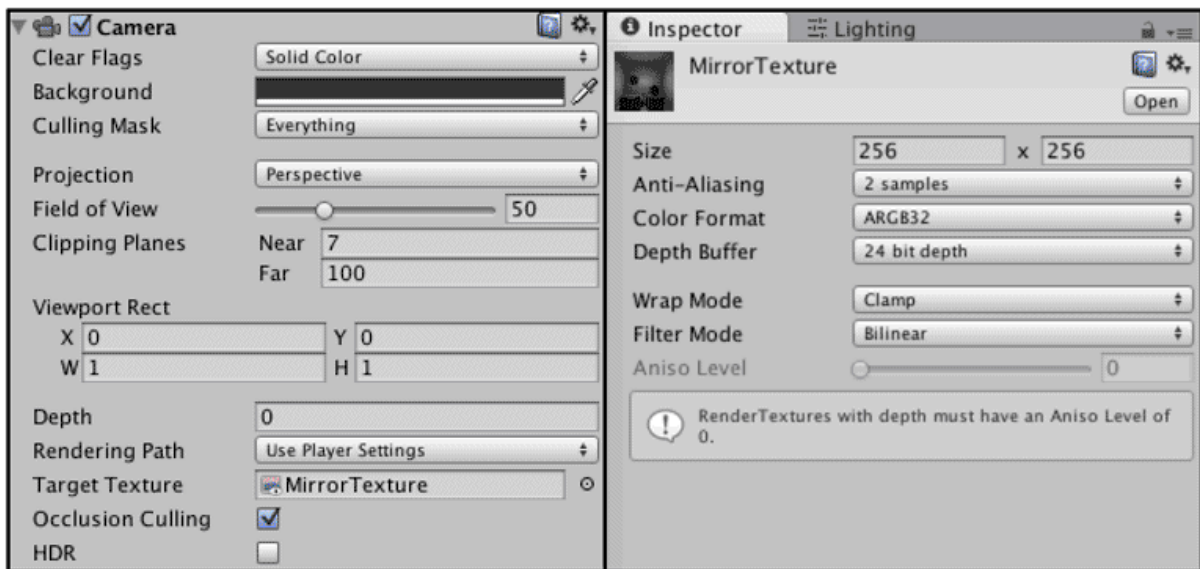
(4) 在场景中创建6个立方体，并调整它们的位置和大小，使得它们构成围绕着摄像机的房间的6面墙。给它们赋予在9.5节中创建的标准材质，并让它们的颜色互不相同。向场景中添加3个点光源，并调整它们的位置，使它们可以照亮整个房间。

(5) 创建3个球体和两个正方体，调整它们的位置和大小，并给它们赋予在9.5节中创建的标准材质。这些物体将作为房间内的饰品。

(6) 创建一个四边形(Quad)，调整它的位置和大小，它将作为镜子。把第2步中创建的材质赋给它。

(7) 在Project视图下创建一个渲染纹理(右键单击Create → Render Texture)，在本书资源中，该渲染纹理名为MirrorTexture。它使用的纹理设置如图10.12右图所示。

(8) 最后，为了得到从镜子出发观察到的场景图像，我们还需要创建一个摄像机，并调整它的位置、裁剪平面、视角等，使得它的显示图像是我们希望的镜子图像。由于这个摄像机不需要直接显示在屏幕上，而是用于渲染到纹理。因此，我们把第7步中创建的MirrorTexture拖曳到该摄像机的Target Texture上。图10.12显示了摄像机面板和渲染纹理的相关设置。



▲图10.12 左图：把摄像机的Target Texture设置成自定义的渲染纹理。右图：渲染纹理使用的纹理设置

镜子实现的原理很简单，它使用一个渲染纹理作为输入属性，并把该渲染纹理在水平方向上翻转后直接显示到物体上即可。打开新建的Chapter10-Mirror，删除所有已有代码，并进行如下关键修改。

(1) 在Properties语义块中声明一个纹理属性，它对应了由镜子摄像机渲染得到的渲染纹理：

```
Properties {
    _MainTex ("Main Tex", 2D) = "white" {}
}
```

(2) 在顶点着色器中计算纹理坐标：

```
v2f vert(a2v v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.uv = v.texcoord;
    // Mirror needs to flip x
```

```
o.uv.x = 1 - o.uv.x;  
  
return o;  
}
```

在上面的代码中，我们翻转了x分量的纹理坐标。这是因为，镜子里显示的图像都是左右相反的。

(3) 在片元着色器中对渲染纹理进行采样和输出：

```
fixed4 frag(v2f i) : SV_Target {  
    return tex2D(_MainTex, i.uv);  
}
```

保存后返回场景，并把 we 创建的 **MirrorTexture** 渲染纹理拖曳到材质的 **Main Tex** 属性中，就可以得到图10.11中的效果。

在上面的实现中，我们把渲染纹理的分辨率大小设置为256×256。有时，这样的分辨率会使图像模糊不清，此时我们可以使用更高的分辨率或更多的抗锯齿采样等。但需要注意的是，更高的分辨率会影响带宽和性能，我们应当尽量使用较小的分辨率。

### 10.2.2 玻璃效果

在Unity中，我们还可以在Unity Shader中使用一种特殊的Pass来完成获取屏幕图像的目的，这就是**GrabPass**。当我们在Shader中定义了一个**GrabPass**后，Unity会把当前屏幕的图像绘制在一张纹理中，以便我们在后续的Pass中访问它。我们通常会使用**GrabPass**来实现诸如玻璃等透明材质的模拟，与使用简单的透明混合不同，使用**GrabPass**可以让我们对该物体后面的图像进行更复杂的处理，例如使用法线来模拟折射效果，而不再是简单的和原屏幕颜色进行混合。

需要注意的是，在使用GrabPass的时候，我们需要额外小心**物体的渲染队列设置**。正如之前所说，GrabPass通常用于渲染透明物体，尽管代码里并不包含混合指令，但我们往往仍然需要把物体的渲染队列设置成透明队列（即"Queue"="Transparent"）。这样才可以保证当渲染该物体时，所有的不透明物体都已经被绘制在屏幕上，从而获取正确的屏幕图像。

在本节中，我们将会使用GrabPass来模拟一个玻璃效果。在学习完本节后，我们可以得到类似图10.13中的效果。这种效果的实现非常简单，我们首先使用一张法线纹理来修改模型的法线信息，然后使用了10.1节介绍的反射方法，通过一个Cubemap来模拟玻璃的反射，而在模拟折射时，则使用了GrabPass获取玻璃后面的屏幕图像，并使用切线空间下的法线对屏幕纹理坐标偏移后，再对屏幕图像进行采样来模拟近似的折射效果。

为此，我们需要做如下准备工作。

（1）新建一个场景。在本书资源中，该场景名为Scene\_10\_2\_2。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window → Lighting → Skybox中去掉场景中的天空盒子。

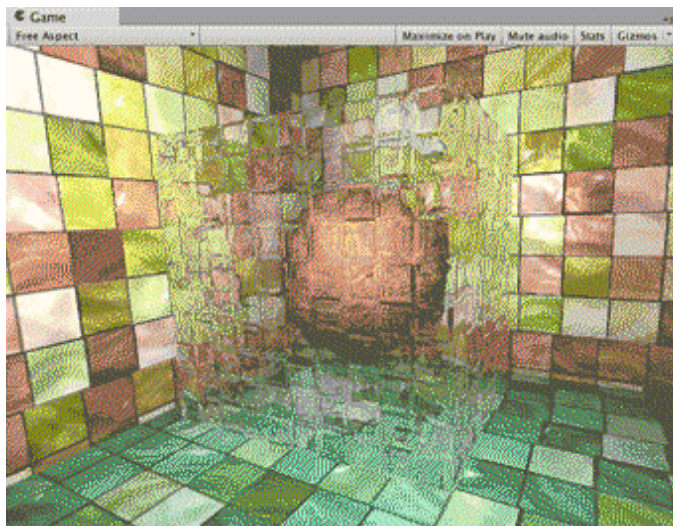
（2）新建一个材质。在本书资源中，该材质名为GlassRefractionMat。

（3）新建一个Unity Shader。在本书资源中，该Shader名为Chapter10-GlassRefraction。把新的Unity Shader赋给第2步中创建的材

质。

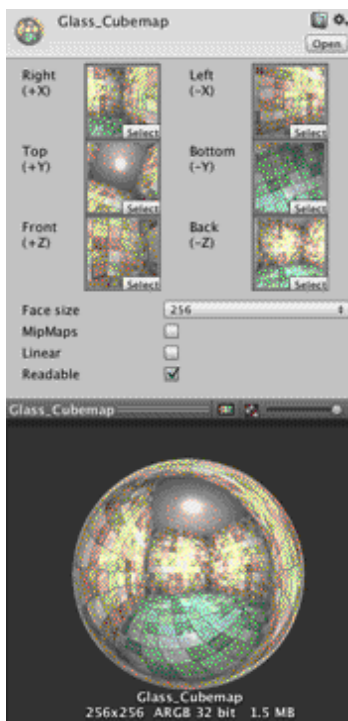
(4) 构建一个测试玻璃效果的场景。在本书资源的实现中，我们构建了一个由6面墙围成的封闭房间，并在房间中放置了一个立方体和一个球体，其中球体位于立方体内部，这是为了模拟玻璃对内部物体的折射效果。把第2步中创建的材质赋给立方体。

(5) 为了得到本场景适用的环境映射纹理，我们使用了10.1.2节中实现的创建立方体纹理的脚本（通过GameObject → Render into Cubemap打开编辑窗口）来创建它，如图10.14所示。在本书资源中，该Cubemap名为Glass\_Cubemap。



▲ 图10.13 玻璃效果





▲ 图10.14 本例使用的立方体纹理

完成准备工作后，打开Chapter10-GlassRefraction，对它进行如下关键修改。

(1) 首先，我们需要声明该Shader使用的各个属性：

```
Properties {
    _MainTex ("Main Tex", 2D) = "white" {}
    _BumpMap ("Normal Map", 2D) = "bump" {}
    _Cubemap ("Environment Cubemap", Cube) = "_Skybox" {}
    _Distortion ("Distortion", Range(0, 100)) = 10
    _RefractAmount ("Refract Amount", Range(0.0, 1.0)) = 1.0
}
```

其中，`_MainTex`是该玻璃的材质纹理，默认为白色纹理；`_BumpMap`是玻璃的法线纹理；`_Cubemap`是用于模拟反射的环境纹理；`_Distortion`则用于控制模拟折射时图像的扭曲程度；`_RefractAmount`用于控制折射程度，当`_RefractAmount`值为0时，该玻

璃只包含反射效果，当\_RefractAmount值为1时，该玻璃只包括折射效果。

(2) 定义相应的渲染队列，并使用GrabPass来获取屏幕图像：

```
SubShader {  
    // We must be transparent, so other objects are drawn before  
    this one.  
    Tags { "Queue"="Transparent" "RenderType"="Opaque" }  
  
    // This pass grabs the screen behind the object into a texture.  
    // We can access the result in the next pass as _RefractionTex  
    GrabPass { "_RefractionTex" }
```

我们首先在SubShader的标签中将渲染队列设置成Transparent，尽管在后面的RenderType被设置为了Opaque。这两者看似矛盾，但实际上服务于不同的需求。我们在之前说过，把Queue设置成Transparent可以确保该物体渲染时，其他所有不透明物体都已经被渲染到屏幕上了，否则就可能无法正确得到“透过玻璃看到的图像”。而设置RenderType则是为了在使用着色器替换（Shader Replacement）时，该物体可以在需要时被正确渲染。这通常发生在我们需要得到摄像机的深度和法线纹理时，这将会在第13章中学到。

随后，我们通过关键词GrabPass定义了一个抓取屏幕图像的Pass。在这个Pass中我们定义了一个字符串，该字符串内部的名称决定了抓取得到的屏幕图像将会被存入哪个纹理中。实际上，我们可以省略声明该字符串，但直接声明纹理名称的方法往往可以得到更高的性能，具体原因可以参见本节最后的部分。

(3) 定义渲染玻璃所需的Pass。为了在Shader中访问各个属性，我们首先需要定义它们对应的变量：

```
sampler2D _MainTex;  
float4 _MainTex_ST;  
sampler2D _BumpMap;  
float4 _BumpMap_ST;  
samplerCUBE _Cubemap;  
float _Distortion;  
fixed _RefractAmount;  
sampler2D _RefractionTex;  
float4 _RefractionTex_TexelSize;
```

需要注意的是，我们还定义了`_RefractionTex`和`_RefractionTex_TexelSize`变量，这对应了在使用`GrabPass`时指定的纹理名称。`_RefractionTex_TexelSize`可以让我们得到该纹理的纹素大小，例如一个大小为 $256 \times 512$ 的纹理，它的纹素大小为 $(1/256, 1/512)$ 。我们需要在对屏幕图像的采样坐标进行偏移时使用该变量。

(4) 我们首先需要定义顶点着色器：

```
v2f vert (a2v v) {  
    v2f o;  
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);  
  
    o.scrPos = ComputeGrabScreenPos(o.pos);  
  
    o.uv.xy = TRANSFORM_TEX(v.texcoord, _MainTex);  
    o.uv.zw = TRANSFORM_TEX(v.texcoord, _BumpMap);  
  
    float3 worldPos = mul(_Object2World, v.vertex).xyz;  
    fixed3 worldNormal = UnityObjectToWorldNormal(v.normal);  
    fixed3 worldTangent = UnityObjectToWorldDir(v.tangent.xyz);  
    fixed3 worldBinormal = cross(worldNormal, worldTangent) *  
v.tangent.w;  
  
    o.TtoW0 = float4(worldTangent.x, worldBinormal.x,  
worldNormal.x, worldPos.x);  
    o.TtoW1 = float4(worldTangent.y, worldBinormal.y,  
worldNormal.y, worldPos.y);  
    o.TtoW2 = float4(worldTangent.z, worldBinormal.z,  
worldNormal.z, worldPos.z);  
  
    return o;  
}
```

在进行了必要的顶点坐标变换后，我们通过调用内置的 `ComputeGrabScreenPos` 函数来得到对应被抓取的屏幕图像的采样坐标。读者可以在 `UnityCG.cginc` 文件中找到它的声明，它的主要代码和 `ComputeScreenPos` 基本类似，最大的不同是针对平台差异造成的采样坐标问题（详见5.6.1节）进行了处理。接着，我们计算了 `_MainTex` 和 `_BumpMap` 的采样坐标，并把它们分别存储在一个 `float4` 类型变量的 `xy` 和 `zw` 分量中。由于我们需要在片元着色器中把法线方向从切线空间（由法线纹理采样得到）变换到世界空间下，以便对 `Cubemap` 进行采样，因此，我们需要在这里计算该顶点对应的从切线空间到世界空间的变换矩阵，并把该矩阵的每一行分别存储在 `TtoW0`、`TtoW1` 和 `TtoW2` 的 `xyz` 分量中。这里面使用的数学方法就是，得到切线空间下的3个坐标轴（`xyz` 轴分别对应了切线、副切线和法线的方向）在世界空间下的表示，再把它们依次按列组成一个变换矩阵即可。`TtoW0` 等值的 `w` 轴同样被利用起来，用于存储世界空间下的顶点坐标。

(5) 然后，定义片元着色器：

```
fixed4 frag (v2f i) : SV_Target {
    float3 worldPos = float3(i.TtoW0.w, i.TtoW1.w, i.TtoW2.w);
    fixed3 worldViewDir =
        normalize(UnityWorldSpaceViewDir(worldPos));

    // Get the normal in tangent space
    fixed3 bump = UnpackNormal(tex2D(_BumpMap, i.uv.zw));

    // Compute the offset in tangent space
    float2 offset = bump.xy * _Distortion *
        _RefractionTex_TexelSize.xy;
    i.scrPos.xy = offset + i.scrPos.xy;
    fixed3 refrCol = tex2D(_RefractionTex,
        i.scrPos.xy/i.scrPos.w).rgb;

    // Convert the normal to world space
    bump = normalize(half3(dot(i.TtoW0.xyz, bump), dot(i.TtoW1.xyz,
        bump), dot(i.TtoW2.xyz, bump)));
```

```
fixed3 reflDir = reflect(-worldViewDir, bump);
fixed4 texColor = tex2D(_MainTex, i.uv.xy);
fixed3 reflCol = texCUBE(_Cubemap, reflDir).rgb * texColor.rgb;

fixed3 finalColor = reflCol * (1 - _RefractAmount) + refrCol *
_RefractAmount;
return fixed4(finalColor, 1);
}
```

我们首先通过TtoW0等变量的w分量得到世界坐标，并用该值得到该片元对应的视角方向。随后，我们对法线纹理进行采样，得到切线空间下的法线方向。我们使用该值和\_RefractAmount属性以及\_RefractAmount来对屏幕图像的采样坐标进行偏移，模拟折射效果。\_Distortion值越大，偏移量越大，玻璃背后的物体看起来变形程度越大。在这里，我们选择使用切线空间下的法线方向来进行偏移，是因为该空间下的法线可以反映顶点局部空间下的法线方向。随后，我们对scrPos透视除法得到真正的屏幕坐标（原理可参见4.9.3节），再使用该坐标对抓取的屏幕图像\_RefractTex进行采样，得到模拟的折射颜色。

之后，我们把法线方向从切线空间变换到了世界空间下（使用变换矩阵的每一行，即TtoW0、TtoW1和TtoW2，分别和法线方向点乘，构成新的法线方向），并据此得到视角方向相对于法线方向的反射方向。随后，使用反射方向对Cubemap进行采样，并把结果和主纹理颜色相乘后得到反射颜色。

最后，我们使用\_RefractAmount属性对反射和折射颜色进行混合，作为最终的输出颜色。

完成后，我们把本书资源中的Glass\_Diffuse.jpg和Glass\_Normal.jpg文件赋给材质的Main Tex和Normal Map属性，把之前创建的Glass\_Cubemap赋给Environment Cubemap属性，再调整\_RefractAmount属性即可得到类似图10.13中的玻璃效果。

在前面的实现中，我们在GrabPass中使用一个字符串指明了被抓取的屏幕图像将会存储在哪个名称的纹理中。实际上，GrabPass支持两种形式。

- 直接使用GrabPass { }，然后在后续的Pass中直接使用GrabTexture来访问屏幕图像。但是，当场景中有多个物体都使用了这样的形式来抓取屏幕时，这种方法的性能消耗比较大，因为对于每一个使用它的物体，Unity都会为它单独进行一次昂贵的屏幕抓取操作。但这种方法可以让每个物体得到不同的屏幕图像，这取决于它们的渲染队列及渲染它们时当前的屏幕缓冲中的颜色。
- 使用GrabPass { "TextureName" }，正如本节中的实现，我们可以在后续的Pass中使用TextureName来访问屏幕图像。使用这种方法同样可以抓取屏幕，但Unity只会在每一帧时为第一个使用名为TextureName的纹理的物体执行一次抓取屏幕的操作，而这个纹理同样可以在其他Pass中被访问。这种方法更高效，因为不管场景中有多少物体使用了该命令，每一帧中Unity都只会执行一次抓取工作，但这也意味着所有物体都会使用同一张屏幕图像。不过，在大多数情况下这已经足够了。

### 10.2.3 渲染纹理 vs. GrabPass

尽管GrabPass和10.2.1节中使用的渲染纹理 + 额外摄像机的方式都可以抓取屏幕图像，但它们之间还是有一些不同的。GrabPass的好处在于实现简单，我们只需要在Shader中写几行代码就可以实现抓取屏幕的目的。而要使用渲染纹理的话，我们首先需要创建一个渲染纹理和一个额外的摄像机，再把该摄像机的Render Target设置为新建的渲染纹理对象，最后把该渲染纹理传递给相应的Shader。

但从效率上来讲，使用渲染纹理的效率往往要好于GrabPass，尤其在移动设备上。使用渲染纹理我们可以自定义渲染纹理的大小，尽管这种方法需要把部分场景再次渲染一遍，但我们可以通过调整摄像机的渲染层来减少二次渲染时的场景大小，或使用其他方法来控制摄像机是否需要开启。而使用GrabPass获取到的图像分辨率和显示屏幕是一致的，这意味着在一些高分辨率的设备上可能会造成严重的带宽影响。而且在移动设备上，GrabPass虽然不会重新渲染场景，但它往往需要CPU直接读取后备缓冲（back buffer）中的数据，破坏了CPU和GPU之间的并行性，这是比较耗时的，甚至在一些移动设备上这是不支持的。

在Unity 5中，Unity引入了**命令缓冲（Command Buffers）**来允许我们扩展Unity的渲染流水线。使用命令缓冲我们也可以得到类似抓屏的效果，它可以在不透明物体渲染后把当前的图像复制到一个临时的渲染目标纹理中，然后在那里进行一些额外的操作，例如模糊等，最后把图像传递给需要使用它的物体进行处理和显示。除此之外，命令缓冲还允许我们实现很多特殊的效果，读者可以在Unity官方手册的**图像命令缓冲**一文（<http://docs.unity3d.com/Manual/Graphics>



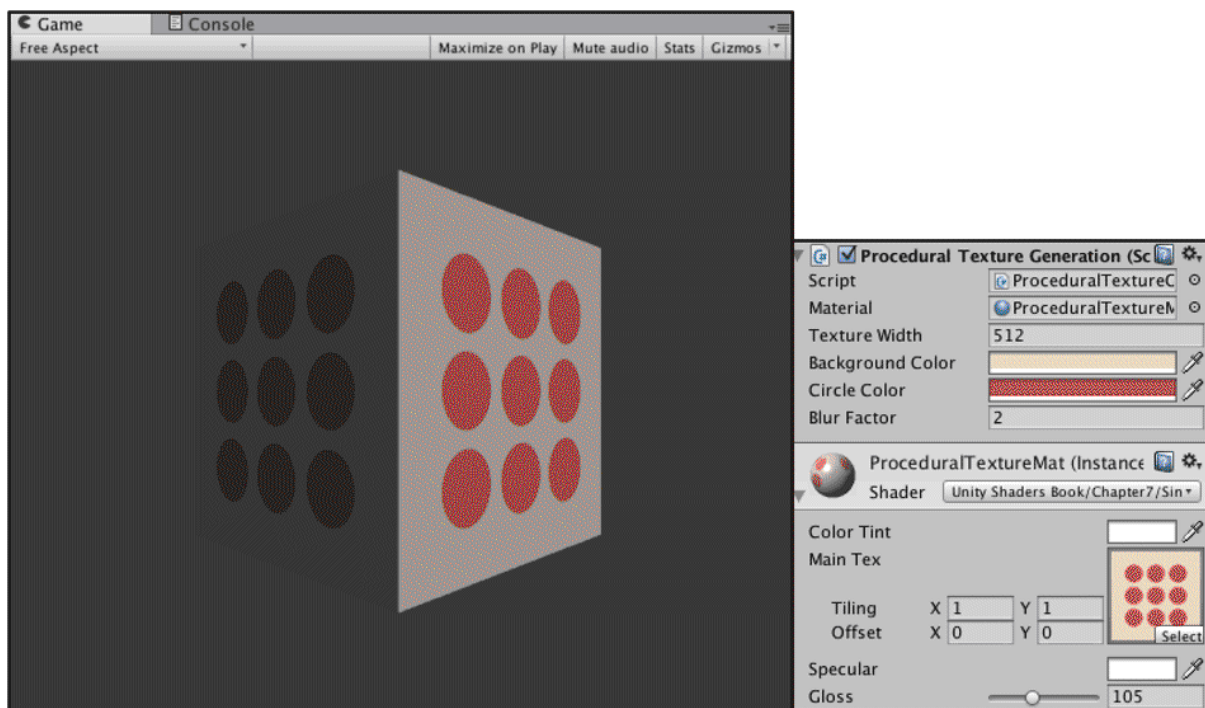
CommandBuffers.html) 中找到更多内容，Unity 还提供了一个示例工程供我们学习。

## 10.3 程序纹理

**程序纹理 (Procedural Texture)** 指的是那些由计算机生成的图像，我们通常使用一些特定的算法来创建个性化图案或非常真实的自然元素，例如木头、石子等。使用程序纹理的好处在于我们可以使用各种参数来控制纹理的外观，而这些属性不仅仅是那些颜色属性，甚至可以是完全不同类型的图案属性，这使得我们可以得到更加丰富的动画和视觉效果。在本节中，我们首先会尝试用算法来实现一个非常简单的程序材质。然后，我们会介绍Unity里一类专门使用程序纹理的材质——程序材质。

### 10.3.1 在Unity中实现简单的程序纹理

在这一节里，我们会使用一个算法来生成一个波点纹理，如图10.15所示。我们可以在脚本中调整一些参数，如背景颜色、波点颜色等，以控制最终生成的纹理外观。



▲ 图10.15 脚本生成的程序纹理

为此，我们需要进行如下准备工作。

(1) 新建一个场景。在本书资源中，该场景名为Scene\_10\_3\_1。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window → Lighting → Skybox中去掉场景中的天空盒子。

(2) 新建一个材质。在本书资源中，该材质名为ProceduralTextureMat。

(3) 我们使用第7章的一个Unity Shader——Chapter7-SingleTexture，把它赋给第2步中创建的材质。

(4) 新建一个立方体，并把第2步中的材质赋给它。

(5) 我们并没有为ProceduralTextureMat材质赋予任何纹理，这是因为，我们想要用脚本来创建程序纹理。为此，我们再创建一个脚本ProceduralTextureGeneration.cs，并把它拖曳到第4步创建的立方体。

在本节中，我们将会使用代码来生成一个波点纹理。为此，我们打开ProceduralTextureGeneration.cs，进行如下修改。

(1) 为了让该脚本能够在编辑器模式下运行，我们首先在类的开头添加如下代码：

```
[ExecuteInEditMode]
public class ProceduralTextureGeneration : MonoBehaviour {
```

(2) 声明一个材质，这个材质将使用该脚本中生成的程序纹理：

```
public Material material = null;
```

(3) 然后，声明该程序纹理使用的各种参数：

```
#region Material properties
[SerializeField, SetProperty("texturewidth")]
private int m_textureWidth = 512;
public int textureWidth {
    get {
        return m_textureWidth;
    }
    set {
        m_textureWidth = value;
        _UpdateMaterial();
    }
}

[SerializeField, SetProperty("backgroundColor")]
private Color m_backgroundColor = Color.white;
public Color backgroundColor {
    get {
        return m_backgroundColor;
    }
    set {
```

```

        m_backgroundColor = value;
        _UpdateMaterial();
    }
}

[SerializeField, SetProperty("circleColor")]
private Color m_circleColor = Color.yellow;
public Color circleColor {
    get {
        return m_circleColor;
    }
    set {
        m_circleColor = value;
        _UpdateMaterial();
    }
}

[SerializeField, SetProperty("blurFactor")]
private float m_blurFactor = 2.0f;
public float blurFactor {
    get {
        return m_blurFactor;
    }
    set {
        m_blurFactor = value;
        _UpdateMaterial();
    }
}
}
#endregion

```

**#region**和**#endregion**仅仅是为了组织代码，并没有其他作用。由于我们生成的纹理是由若干圆点构成的，因此在上面的代码中，我们声明了4个纹理属性：纹理的大小，数值通常是2的整数幂；纹理的背景颜色；圆点的颜色；模糊因子，这个参数是用来模糊圆形边界的。注意到，对于每个属性我们使用了**get/set**的方法，为了在面板上修改属性时仍可以执行**set**函数，我们使用了一个开源插件**SetProperty**

（<https://github.com/LMNRY/SetProperty/blob/master/Scripts/SetPropertyExample.cs>）。这使得当我们修改了材质属性时，可以执行**\_UpdateMaterial**函数来使用新的属性重新生成程序纹理。

(4) 为了保存生成的程序纹理，我们声明一个Texture2D类型的纹理变量：

```
private Texture2D m_generatedTexture = null;
```

(5) 下面开始编写各个函数。首先，我们需要在Start函数中进行相应的检查，以得到需要使用该程序纹理的材质：

```
void Start () {  
    if (material == null) {  
        Renderer renderer = gameObject.GetComponent<Renderer>();  
        if (renderer == null) {  
            Debug.LogWarning("Cannot find a renderer.");  
            return;  
        }  
  
        material = renderer.sharedMaterial;  
    }  
  
    _UpdateMaterial();  
}
```

在上面的代码里，我们首先检查了material变量是否为空，如果为空，就尝试从使用该脚本所在的物体上得到相应的材质。完成后，调用\_UpdateMaterial函数来为其生成程序纹理。

(6) \_UpdateMaterial函数的代码如下：

```
private void _UpdateMaterial() {  
    if (material != null) {  
        m_generatedTexture = _GenerateProceduralTexture();  
        material.SetTexture("_MainTex", m_generatedTexture);  
    }  
}
```

它确保material不为空，然后调用\_GenerateProceduralTexture函数来生成一张程序纹理，并赋给m\_generatedTexture变量。完成后，利用

Material.SetTexture函数把生成的纹理赋给材质。材质material中需要有一个名为\_MainTex的纹理属性。

(7) \_GenerateProceduralTexture函数的代码如下:

```
private Texture2D _GenerateProceduralTexture() {
    Texture2D proceduralTexture = new Texture2D(textureWidth,
textureWidth);

    // 定义圆与圆之间的间距
    float circleInterval = textureWidth / 4.0f;
    // 定义圆的半径
    float radius = textureWidth / 10.0f;
    // 定义模糊系数
    float edgeBlur = 1.0f / blurFactor;

    for (int w = 0; w < textureWidth; w++) {
        for (int h = 0; h < textureWidth; h++) {
            // 使用背景颜色进行初始化
            Color pixel = backgroundColor;

            // 依次画9个圆
            for (int i = 0; i < 3; i++) {
                for (int j = 0; j < 3; j++) {
                    // 计算当前所绘制的圆的圆心位置
                    Vector2 circleCenter = new
Vector2(circleInterval * (i + 1), circleInterval
* (j + 1));

                    // 计算当前像素与圆心的距离
                    float dist = Vector2.Distance(new Vector2(w,
h), circleCenter) - radius;

                    // 模糊圆的边界
                    Color color = _MixColor(circleColor, new
Color(pixel.r, pixel.g,
pixel.b, 0.0f), Mathf.SmoothStep(0f, 1.0f, dist
* edgeBlur));

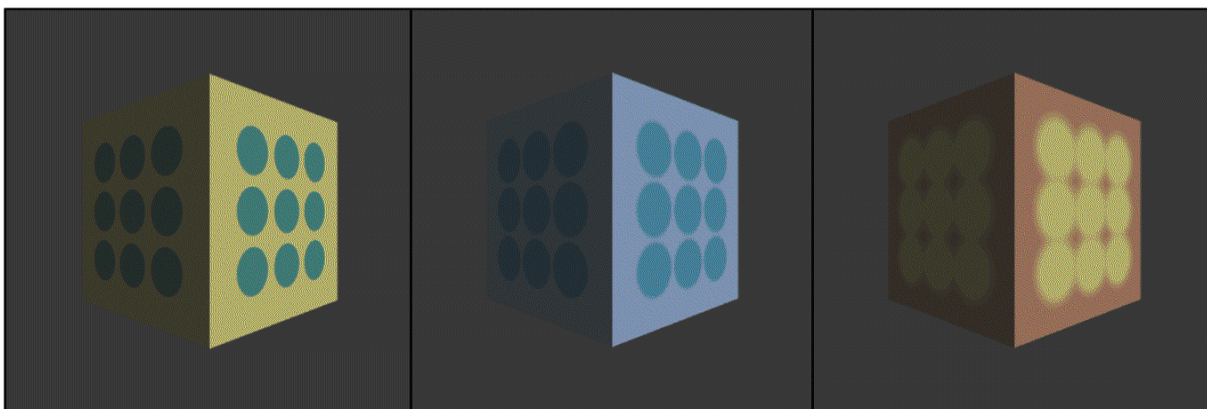
                    // 与之前得到的颜色进行混合
                    pixel = _MixColor(pixel, color, color.a);
                }
            }

            proceduralTexture.SetPixel(w, h, pixel);
        }
    }
}
```

```
proceduralTexture.Apply();  
return proceduralTexture;  
}
```

代码首先初始化一张二维纹理，并且提前计算了一些生成纹理时需要的变量。然后，使用了一个两层的嵌套循环遍历纹理中的每个像素，并在纹理上依次绘制9个圆形。最后，调用Texture2D.Apply函数来强制把像素值写入纹理中，并返回该程序纹理。

保存脚本后返回场景，调整相应的参数后可以得到类似图10.15中的效果。我们可以调整脚本面板中的材质参数来得到不同的程序纹理，如图10.16所示。



▲ 图10.16 调整程序纹理的参数来得到不同的程序纹理

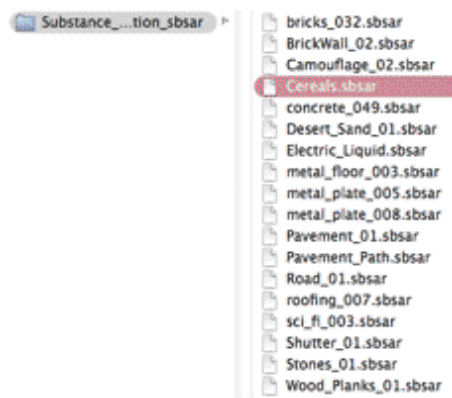
至此，我们已经学会如何通过脚本来创建一个程序纹理，再赋给相应的材质了。

### 10.3.2 Unity的程序材质



在Unity中，有一类专门使用程序纹理的材质，叫做**程序材质（Procedural Materials）**。这类材质和我们之前使用的那些材质在本质上是一样的，不同的是，它们使用的纹理不是普通的纹理，而是程序纹理。需要注意的是，程序材质和它使用的程序纹理并不是在Unity中创建的，而是使用了一个名为**Substance Designer**的软件在Unity外部生成的。

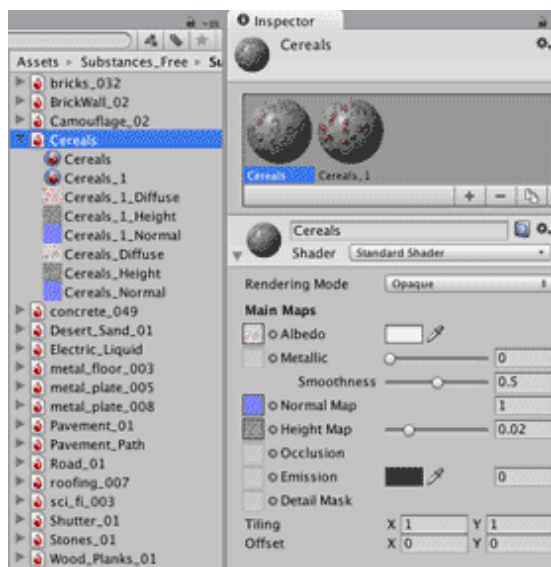
**Substance Designer**是一个非常出色的纹理生成工具，很多3A的游戏项目都使用了由它生成的材质。我们可以从Unity的资源商店或网络中获取到很多免费或付费的Substance材质。这些材质都是以.sbsar为后缀的，如图10.17所示（资源来源于<https://www.assetstore.unity3d.com/en/#!/content/1352>）。我们可以直接把这些材质像其他资源一样拖入Unity项目中。



▲ 图10.17 后缀为.sbsar的Substance材质

当把这些文件导入Unity后，Unity就会生成一个**程序纹理资源（Procedural Material Asset）**。程序纹理资源可以包含一个或多个程序材质，例如图10.18中就包含了两个程序纹理——Cereals和

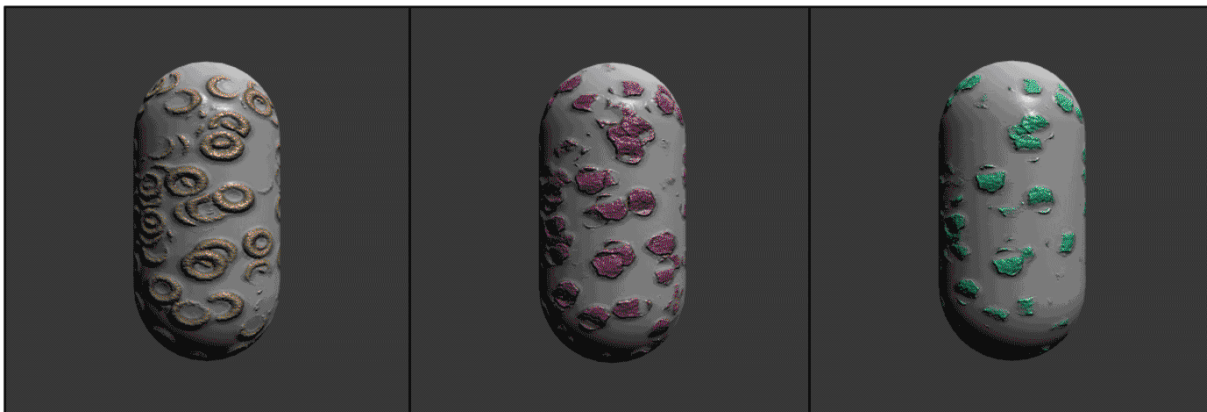
Cereals\_1，每个程序纹理使用了不同的纹理参数，因此Unity为它们生成了不同的程序纹理，例如Cereals\_Diffuse和Cereals\_1\_Diffuse等。



▲ 图10.18 程序纹理资源

通过单击程序材质，我们可以在程序纹理的面板上看到该材质使用的Unity Shader及其属性、生成程序纹理使用的纹理属性、材质预览等信息。

程序材质的使用和普通材质是一样的，我们把它们拖曳到相应的模型上即可。读者可以在本书资源的Scene\_10\_3\_2中找到这样的示例场景。程序纹理的强大之处很大原因在于它的多变性，我们可以通过调整程序纹理的属性来控制纹理的外观，甚至可以生成看似完全不同的纹理。图10.19给出了调整Cereals程序材质的不同纹理属性得到的不同材质效果。



▲图10.19 调整程序纹理属性可以得到看似完全不同的程序材质效果

可以看出，程序材质的自由度很高，而且可以和Shader配合得到非常出色的视觉效果，它是一种非常强大的材质类型。

# 第11章 让画面动起来

没有动画的画面往往让人觉得很无趣。在本章中，我们将会学习如何向Unity Shader中引入时间变量，以实现各种动画效果。在11.1节中，我们首先会介绍Unity Shader内置的时间变量，在随后的章节中我们会使用这些时间变量来实现动画。11.2节会介绍两种常见的纹理动画，即序列帧动画和背景循环滚动动画。在11.3节，我们会学习使用顶点动画来实现流动的河流、广告牌等动画效果，并在最后给出一些在实现顶点动画时的注意事项。

## 11.1 Unity Shader中的内置变量（时间篇）

动画效果往往都是把时间添加到一些变量的计算中，以便在时间变化时画面也可以随之变化。Unity Shader提供了一系列关于时间的内置变量来允许我们方便地在Shader中访问运行时间，实现各种动画效果。表11.1给出了这些内置的时间变量。

表11.1 Unity内置的时间变量

名 称	类 型	描 述
_Time	float4	t是自该场景加载开始所经过的时间，4个分量的值分别是(t/20, t, 2t, 3t)。

名 称	类 型	描 述
_SinTime	float4	t是时间的正弦值，4个分量的值分别是(t/8, t/4, t/2, t)
_CosTime	float4	t是时间的余弦值，4个分量的值分别是(t/8, t/4, t/2, t)
unity_DeltaTime	float4	dt是时间增量，4个分量的值分别是(dt, 1/dt, smoothDt, 1/smoothDt)

在后面的章节中，我们会使用上述时间变量来实现纹理动画和顶点动画。

## 11.2 纹理动画

纹理动画在游戏中的应用非常广泛。尤其在各种资源都比较局限的移动平台上，我们往往会使用纹理动画来代替复杂的粒子系统等模拟各种动画效果。

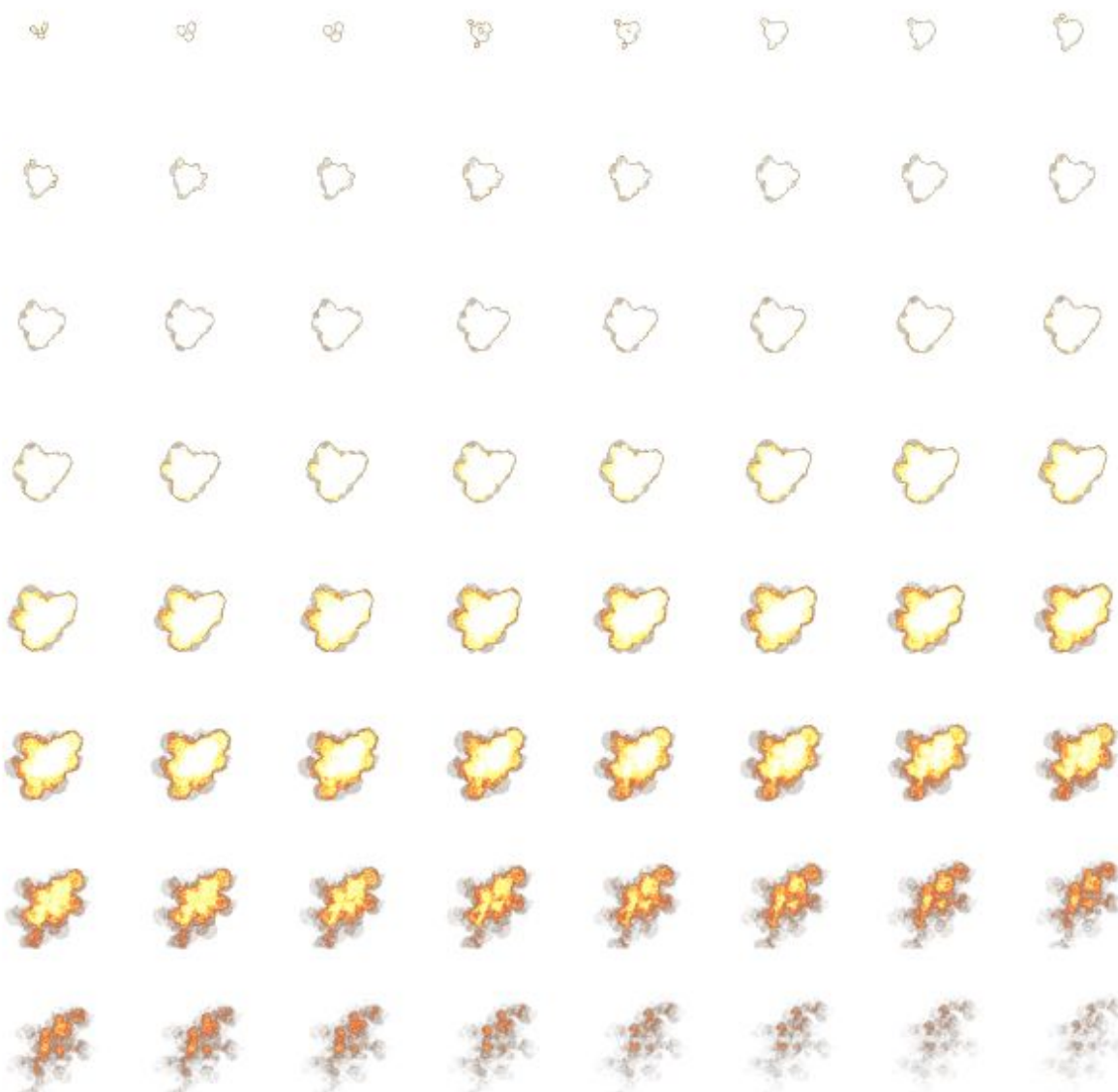
### 11.2.1 序列帧动画

最常见的纹理动画之一就是序列帧动画。序列帧动画的原理非常简单，它像放电影一样，依次播放一系列关键帧图像，当播放速度达到一定数值时，看起来就是一个连续的动画。它的优点在于灵活性很强，我们不需要进行任何物理计算就可以得到非常细腻的动物效果。

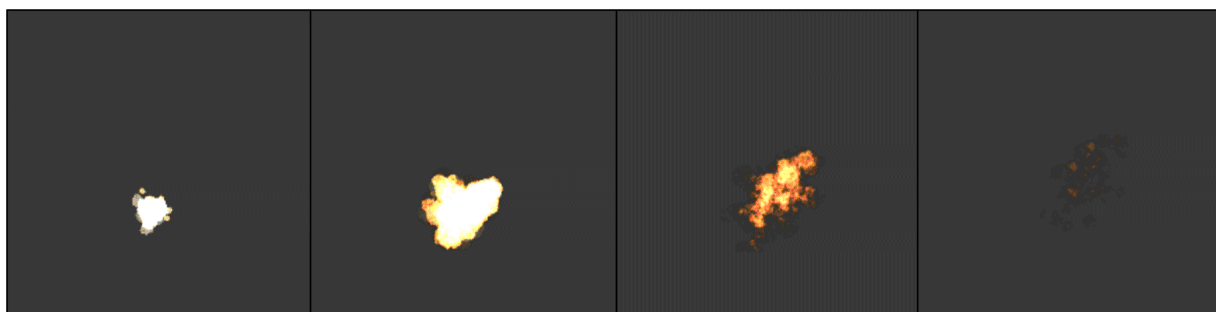
而它的缺点也很明显，由于序列帧中每张关键帧图像都不一样，因此，要制作一张出色的序列帧纹理所需要的美术工程量也比较大。

要想实现序列帧动画，我们先要提供一张包含了关键帧图像的图像。在本书资源中，我们提供了这样一张图像（Assets/Textures/Chapter11/Boom.png），如图11.1所示。

上述图像包含了 $8 \times 8$ 张关键帧图像，它们的大小相同，而且播放顺序为从左到右、从上到下。图11.2给出了不同时刻播放的不同动画效果。



▲图11.1 本节使用的序列帧图像





▲图11.2 使用序列帧动画来实现爆炸效果

为了在Unity中实现序列帧动画，我们需要做如下准备工作。

(1) 新建一个场景。在本书资源中，该场景名为Scene\_11\_2\_1。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window → Lighting → Skybox中去掉场景中的天空盒子。

(2) 新建一个材质。在本书资源中，该材质名为ImageSequenceAnimationMat。

(3) 新建一个Unity Shader。在本书资源中，该Shader名为Chapter11-ImageSequenceAnimation。把新的Shader赋给第2步中创建的材质。

(4) 在场景中创建一个四边形（Quad），调整它的位置使其正面朝向摄像机，并把第2步中的材质拖给曳它。

上述序列帧动画的精髓在于，我们需要在每个时刻计算该时刻下应该播放的关键帧的位置，并对该关键帧进行纹理采样。打开新建的Chapter11-ImageSequenceAnimation，删除原有的代码，并添加如下关键代码。

(1) 我们首先声明了多个属性，以设置该序列帧动画的相关参数：

```
Properties {  
    _Color ("Color Tint", Color) = (1, 1, 1, 1)  
    _MainTex ("Image Sequence", 2D) = "white" {}  
    _HorizontalAmount ("Horizontal Amount", Float) = 4
```

```
_VerticalAmount ("Vertical Amount", Float) = 4
_Speed ("Speed", Range(1, 100)) = 30
}
```

`_MainTex`就是包含了所有关键帧图像的纹理。`_HorizontalAmount`和`_VerticalAmount`分别代表了该图像在水平方向和竖直方向包含的关键帧图像的个数。而`_Speed`属性用于控制序列帧动画的播放速度。

(2) 由于序列帧图像通常是透明纹理，我们需要设置`Pass`的相关状态，以渲染透明效果：

```
SubShader {
    Tags { "Queue"="Transparent" "IgnoreProjector"="True"
    "RenderType"="Transparent" }

    Pass {
        Tags { "LightMode"="ForwardBase" }

        ZWrite Off
        Blend SrcAlpha OneMinusSrcAlpha
    }
}
```

由于序列帧图像通常包含了透明通道，因此可以被当成是一个半透明对象。在这里我们使用半透明的“标配”来设置它的`SubShader`标签，即把`Queue`和`RenderType`设置成`Transparent`，把`IgnoreProjector`设置为`True`。在`Pass`中，我们使用 `Blend`命令来开启并设置混合模式，同时关闭了深度写入。

(3) 顶点着色器的代码非常简单，我们进行了基本的顶点变换，并把顶点纹理坐标存储到了`v2f`结构体里：

```
v2f vert (a2v v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);
    o.uv = TRANSFORM_TEX(v.texcoord, _MainTex);
    return o;
}
```

#### (4) 片元着色器是我们的重头戏:

```
fixed4 frag (v2f i) : SV_Target {
    float time = floor(_Time.y * _Speed);
    float row = floor(time / _HorizontalAmount);
    float column = time - row * _HorizontalAmount;

    // half2 uv = float2(i.uv.x / _HorizontalAmount, i.uv.y /
    // _VerticalAmount);
    // uv.x += column / _HorizontalAmount;
    // uv.y -= row / _VerticalAmount;
    half2 uv = i.uv + half2(column, -row);
    uv.x /= _HorizontalAmount;
    uv.y /= _VerticalAmount;

    fixed4 c = tex2D(_MainTex, uv);
    c.rgb *= _Color;

    return c;
}
```

要播放帧动画，从本质来说，我们需要计算出每个时刻需要播放的关键帧在纹理中的位置。而由于序列帧纹理都是按行按列排列的，因此这个位置可以认为是该关键帧所在的行列索引数。因此，在上面的代码的前3行中我们计算了行列数，其中使用了Unity的内置时间变量 `_Time`。由11.1节可以知道，`_Time.y`就是自该场景加载后所经过的时间。我们首先把 `_Time.y`和速度属性 `_Speed`相乘来得到模拟的时间，并使用CG的 `floor`函数对结果值取整来得到整数时间 `time`。然后，我们使用 `time`除以 `_HorizontalAmount`的结果值的商来作为当前对应的行索引，除法结果的余数则是列索引。接下来，我们需要使用行列索引值来构建真正的采样坐标。由于序列帧图像包含了许多关键帧图像，这意味着采样坐标需要映射到每个关键帧图像的坐标范围内。我们可以首先把原纹理坐标 `i.uv`按行数和列数进行等分，得到每个子图像的纹理坐标范围。然后，我们需要使用当前的行列数对上面的结果进行偏移，得到当前子图像的纹理坐标。需要注意的是，对竖直方向的坐标偏移需

要使用减法，这是因为在Unity中纹理坐标垂直方向的顺序（从下到上逐渐增大）和序列帧纹理中的顺序（播放顺序是从上到下）是相反的。这对应了上面代码中注释掉的代码部分。我们可以把上述过程中的除法整合到一起，就得到了注释下方的代码。这样，我们就得到了真正的纹理采样坐标。

（5）最后，我们把Fallback设置为内置的Transparent/VertexLit（也可以选择关闭Fallback）：

```
Fallback "Transparent/VertexLit"
```

保存后返回场景，我们将Assets/Textures/Chapter11/Boom.png（注意，由于是透明纹理，因此需要勾选该纹理的Alpha Is Transparency属性）赋给ImageSequenceAnimationMat中的Image Sequence属性，并将Horizontal Amount和Vertical Amount设置为8（因为Boom.png包含了8行8列的关键帧图像），完成后单击播放，并调整Speed属性，就可以得到一段连续的爆炸动画。

### 11.2.2 滚动的背景

很多2D游戏都使用了不断滚动的背景来模拟游戏角色在场景中的穿梭，这些背景往往包含了多个层（layers）来模拟一种视差效果。而这些背景的实现往往就是利用了纹理动画。在本节中，我们将实现一个包含了两层的无限滚动的2D游戏背景。本节使用的纹理资源均来自OpenGameArt（<http://opengameart.org>）网站。在学习完本节后，我们可以得到类似图11.3中的效果。单击运行后，就可以得到一个无限滚动的背景效果。



▲ 图11.3 无限滚动的背景（纹理来源： forest-background © 2012-2013 Julien Jorge  
julien.jorge@stuff-o-matic.com）

为此，我们需要进行如下准备工作。

（1）新建一个场景，在本书资源中，该场景名为Scene\_11\_2\_2。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window → Lighting → Skybox中去掉场景中的天空盒子。由于本例模拟的是2D游戏中的滚动背景，因此我们需要把摄像机的投影模式设置为正交投影。

（2）新建一个材质。在本书资源中，该材质名为ScrollingBackgroundMat。

（3）新建一个Unity Shader。在本书资源中，该Shader名为Chapter11-ScrollingBackground。把新的Shader赋给第2步中创建的材质。

(4) 在场景中创建一个四边形 (Quad)，调整它的位置和大小，使它充满摄像机的视野范围，然后把第2步中的材质拖曳给它。该四边形将用于显示游戏背景。

打开新建的Chapter11-ScrollingBackground，删除原有的代码，并添加如下关键代码。

(1) 我们首先声明了新的属性：

```
Properties {  
    _MainTex ("Base Layer (RGB)", 2D) = "white" {}  
    _DetailTex ("2nd Layer (RGB)", 2D) = "white" {}  
    _ScrollX ("Base layer Scroll Speed", Float) = 1.0  
    _Scroll2X ("2nd layer Scroll Speed", Float) = 1.0  
    _Multiplier ("Layer Multiplier", Float) = 1  
}
```

其中，\_MainTex和\_DetailTex分别是第一层（较远）和第二层（较近）的背景纹理，而\_ScrollX和\_Scroll2X对应了各自的水平滚动速度。\_Multiplier参数则用于控制纹理的整体亮度。

(2) 我们的顶点着色器代码非常简单：

```
v2f vert (a2v v) {  
    v2f o;  
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);  
  
    o.uv.xy = TRANSFORM_TEX(v.texcoord, _MainTex) +  
    frac(float2(_ScrollX, 0.0) * _Time.y);  
    o.uv.zw = TRANSFORM_TEX(v.texcoord, _DetailTex) +  
    frac(float2(_Scroll2X, 0.0) * _Time.y);  
  
    return o;  
}
```

我们首先进行了最基本的顶点变换，把顶点从模型空间变换到裁剪空间中。然后，我们计算了两层背景纹理的纹理坐标。为此，我们首先利用**TRANSFORM\_TEX**来得到初始的纹理坐标。然后，我们利用内置的**\_Time.y**变量在水平方向上对纹理坐标进行偏移，以此达到滚动的效果。我们把两张纹理的纹理坐标存储在同一个变量**o.uv**中，以减少占用的插值寄存器空间。

(3) 片元着色器的工作就相对比较简单：

```
fixed4 frag (v2f i) : SV_Target {
    fixed4 firstLayer = tex2D(_MainTex, i.uv.xy);
    fixed4 secondLayer = tex2D(_DetailTex, i.uv.zw);

    fixed4 c = lerp(firstLayer, secondLayer, secondLayer.a);
    c.rgb *= _Multiplier;

    return c;
}
```

我们首先分别利用**i.uv.xy**和**i.uv.zw**对两张背景纹理进行采样。然后，使用第二层纹理的透明通道来混合两张纹理，这使用了CG的**lerp**函数。最后，我们使用**\_Multiplier**参数和输出颜色进行相乘，以调整背景亮度。

(4) 最后，我们把**Fallback**设置为内置的**VertexLit**（也可以选择关闭**Fallback**）：

```
Fallback "VertexLit"
```

保存后返回场景，把本书资源中的  
**Assets/Textures/Chapter11/Far\_Background.png**和  
**Assets/Textures/Chapter11/Near\_Background.png**分别赋给材质的Base



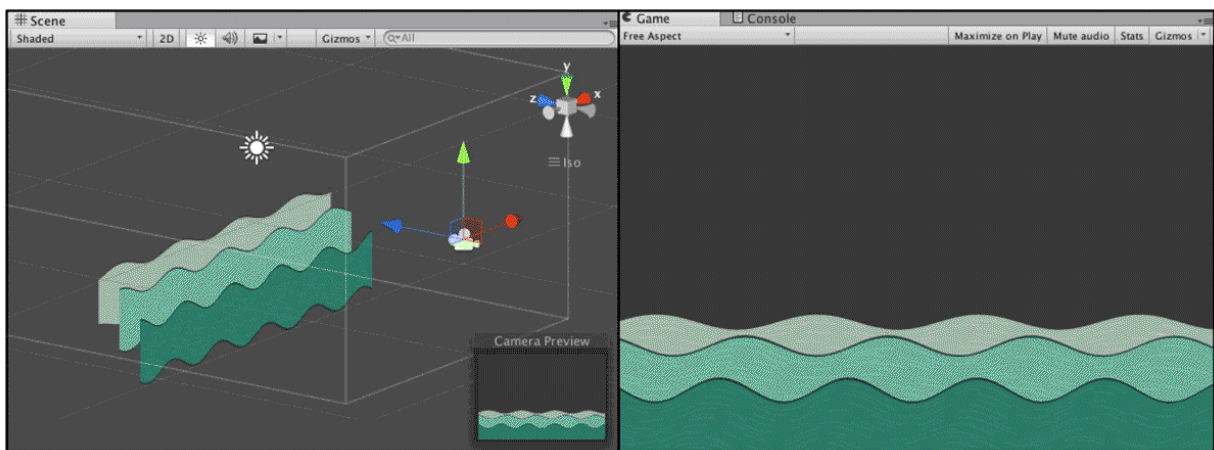
Layer和2nd Layer属性，并调整它们的滚动速度（由于我们想要在视觉上模拟Base Layer比2nd Layer更远的效果，因此Base Layer的滚动速度要比2nd Layer的速度慢一些）。单击运行后，就可以得到类似图11.3中的效果。

## 11.3 顶点动画

如果一个游戏中所有的物体都是静止的，这样枯燥的世界恐怕很难引起玩家的兴趣。顶点动画可以让我们的场景变得更加生动有趣。在游戏中，我们常常使用顶点动画来模拟飘动的旗帜、湍流的小溪等效果。在本节中，我们将学习两种常见的顶点动画的应用——流动的河流以及广告牌技术。在本节最后，我们还将给出一些顶点动画中的注意事项及解决方法。

### 11.3.1 流动的河流

河流的模拟是顶点动画最常见的应用之一。它的原理通常就是使用正弦函数等来模拟水流的波动效果。在本小节中，我们将学习如何模拟一个2D的河流效果。在学习完本节后，我们可以得到类似图11.4中的效果。当单击运行后，可以观察到河流不断流动的效果。



▲ 图11.4 使用顶点动画来模拟2D的河流

为此，我们需要进行如下准备工作。

(1) 新建一个场景。在本书资源中，该场景名为**Scene\_11\_3\_1**。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在**Window → Lighting → Skybox**中去掉场景中的天空盒子。由于本节模拟的是2D效果，因此我们需要把摄像机的投影类型设置为正交投影。

(2) 新建一个材质。在本书资源中，该材质名为**WaterMat**。由于本例需要模拟多层水流效果，我们还创建了**WaterMat1**和**WaterMat2**材质。

(3) 新建一个Unity Shader。在本书资源中，该Shader名为**Chapter11-Water**。把新的Shader赋给第2步中创建的材质。

(4) 在场景中创建多个**Water**模型，调整它们的位置、大小和方向，然后把第2步中的材质拖曳给它们。

打开新建的Chapter11-Water，删除原来的代码，并添加如下关键代码。

(1) 首先，我们声明了一些新的属性：

```
Properties {  
    _MainTex ("Main Tex", 2D) = "white" {}  
    _Color ("Color Tint", Color) = (1, 1, 1, 1)  
    _Magnitude ("Distortion Magnitude", Float) = 1  
    _Frequency ("Distortion Frequency", Float) = 1  
    _InvWaveLength ("Distortion Inverse Wave Length", Float) = 10  
    _Speed ("Speed", Float) = 0.5  
}
```

其中，\_MainTex是河流纹理，\_Color用于控制整体颜色，\_Magnitude用于控制水流波动的幅度，\_Frequency用于控制波动频率，\_InvWaveLength用于控制波长的倒数（\_InvWaveLength越大，波长越小），\_Speed用于控制河流纹理的移动速度。

(2) 在本例中，我们需要为透明效果设置合适的SubShader标签：

```
SubShader {  
    // Need to disable batching because of the vertex animation  
    Tags {"Queue"="Transparent" "IgnoreProjector"="True"  
"RenderType"="Transparent" "DisableBatching"="True"}
```

在上面的设置中，我们除了为透明效果设置Queue、IgnoreProjector和RenderType外，还设置了一个新的标签——**DisableBatching**。我们在3.3.3节中介绍过该标签的含义：一些SubShader在使用Unity的批处理功能时会出现问题，这时可以通过该标签来直接指明是否对该SubShader使用批处理。而这些需要特殊处理的Shader通常就是指包含了模型空间的顶点动画的Shader。这是因为，批处理会合并所有相关的模型，而这些模型各自的模型空间就会丢失。而在本例中，我们需要在物体的模

型空间下对顶点位置进行偏移。因此，在这里需要取消对该Shader的批处理操作。

(3) 接着，我们设置了Pass的渲染状态：

```
Pass {  
    Tags { "LightMode"="ForwardBase" }  
  
    ZWrite Off  
    Blend SrcAlpha OneMinusSrcAlpha  
    Cull Off
```

这里关闭了深度写入，开启并设置了混合模式，并关闭了剔除功能。这是为了让水流的每个面都能显示。

(4) 然后，我们在顶点着色器中进行了相关的顶点动画：

```
v2f vert(a2v v) {  
    v2f o;  
  
    float4 offset;  
    offset.yzw = float3(0.0, 0.0, 0.0);  
    offset.x = sin(_Frequency * _Time.y + v.vertex.x *  
_InvWaveLength + v.vertex.y * _InvWaveLength + v.vertex.z *  
_InvWaveLength) * _Magnitude;  
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex + offset);  
  
    o.uv = TRANSFORM_TEX(v.texcoord, _MainTex);  
    o.uv += float2(0.0, _Time.y * _Speed);  
  
    return o;  
}
```

我们首先计算顶点位移量。我们只希望对顶点的x方向进行位移，因此yzw的位移量被设置为0。然后，我们利用\_Frequency属性和内置的\_Time.y变量来控制正弦函数的频率。为了让不同位置具有不同的位移，我们对上述结果加上了模型空间下的位置分量，并乘以\_InvWaveLength来控制波长。最后，我们对结果值乘以\_Magnitude属性

来控制波动幅度，得到最终的位移。剩下的工作，我们只需要把位移量添加到顶点位置上，再进行正常的顶点变换即可。

在上面的代码中，我们还进行了纹理动画，即使用 `_Time.y` 和 `_Speed` 来控制水平方向上的纹理动画。

(5) 片元着色器的代码非常简单，我们只需要对纹理采样再添加颜色控制即可：

```
fixed4 frag(v2f i) : SV_Target {  
    fixed4 c = tex2D(_MainTex, i.uv);  
    c.rgb *= _Color.rgb;  
  
    return c;  
}
```

(6) 最后，我们把 `Fallback` 设置为内置的 `Transparent/VertexLit`（也可以选择关闭 `Fallback`）：

```
Fallback "Transparent/VertexLit"
```

保存后返回场景，把 `Assets/Textures/Chapter11/Water.psd` 拖曳到材质的 `Main Tex` 属性上，并调整相关参数。为了让河流更加美观，我们可以复制多个材质并使用不同的参数，再赋给不同的 `Water` 模型，就可以得到类似图 11.4 中的效果。

### 11.3.2 广告牌

另一种常见的顶点动画就是**广告牌技术**（**Billboarding**）。广告牌技术会根据视角方向来旋转一个被纹理着色的多边形（通常就是简单的四边形，这个多边形就是广告牌），使得多边形看起来好像总是

面对着摄像机。广告牌技术被用于很多应用，比如渲染烟雾、云朵、闪光效果等。

广告牌技术的本质就是构建旋转矩阵，而我们知道一个变换矩阵需要3个基向量。广告牌技术使用的基向量通常就是**表面法线**

**(normal)**、**指向上的方向 (up)** 以及**指向右的方向 (right)**。除此之外，我们还需要指定一个**锚点 (anchor location)**，这个锚点在旋转过程中是固定不变的，以此来确定多边形在空间中的位置。

广告牌技术的难点在于，如何根据需求来构建3个相互正交的基向量。计算过程通常是，我们首先会通过初始计算得到目标的表面法线（例如就是视角方向）和指向上的方向，而两者往往是不垂直的。但是，两者其中之一是固定的，例如当模拟草丛时，我们希望广告牌的指向上的方向永远是(0, 1, 0)，而法线方向应该随视角变化；而当模拟粒子效果时，我们希望广告牌的法线方向是固定的，即总是指向视角方向，指向上的方向则可以发生变化。我们假设法线方向是固定的，首先，我们根据初始的表面法线和指向上的方向来计算出目标方向的指向右的方向（通过叉积操作）：

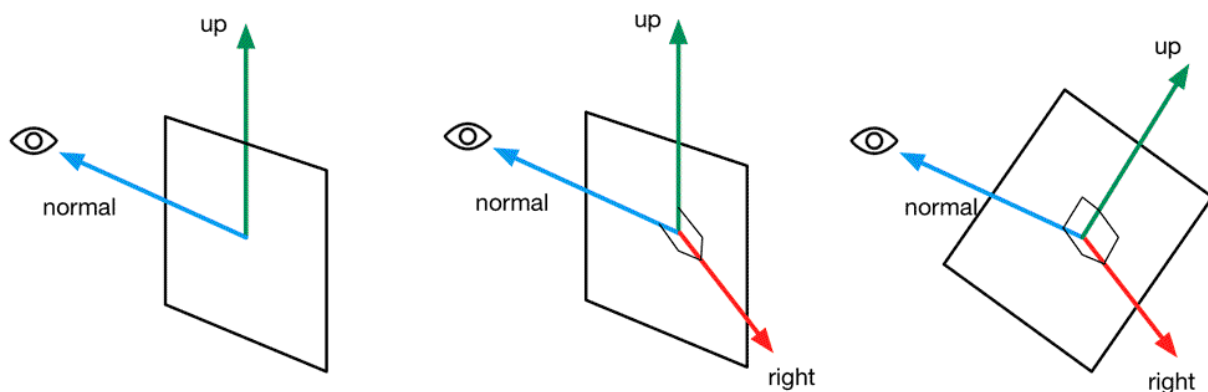
$$\text{right} = \text{up} \times \text{normal}$$

对其归一化后，再由法线方向和指向右的方向计算出正交的指向上的方向即可：

$$\text{up}' = \text{normal} \times \text{right}$$

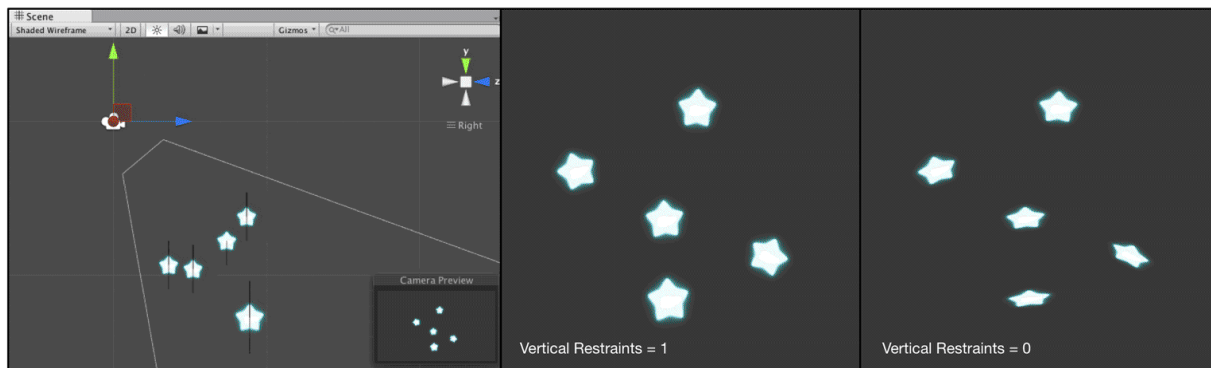
至此，我们就可以得到用于旋转的3个正交基了。图11.5给出了上述计算过程的图示。如果指向上的方向是固定的，计算过程也是类似

的。



▲图11.5 法线固定（总是指向视角方向）时，计算广告牌技术中的3个正交基的过程

下面，我们将在Unity中实现上面提到的广告牌技术。在学习完本节后，我们可以得到类似图11.6中的效果。



▲图11.6 广告牌效果。左图显示了摄像机和5个广告牌之间的位置关系，摄像机是从斜上方向下观察它们的。中间的图显示了当**Vertical Restraints**属性为1，即固定法线方向为观察视角时所得到的效果，可以看出，所有的广告牌都完全面朝摄像机。右图显示了当**Vertical Restraints**属性为0，即固定指向上的方向为 $(0, 1, 0)$ 时所得到的效果，可以看出，广告牌虽然最大限度地面朝摄像机，但其指向上的方向并未发生改变

为此，我们需要进行如下准备工作。



(1) 新建一个场景。在本书资源中，该场景名为Scene\_11\_3\_2。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window → Lighting → Skybox中去掉场景中的天空盒子。

(2) 新建一个材质。在本书资源中，该材质名为BillboardMat。

(3) 新建一个Unity Shader。在本书资源中，该Shader名为Chapter11-Billboard。把新的Shader赋给第2步中创建的材质。

(4) 在场景中创建多个四边形（Quad），调整它们的位置和大小，然后把第2步中的材质拖曳给它们。这些四边形就是用于广告牌技术的广告牌。

打开新建的Chapter11-Billboard，删除原有的代码，添加如下关键代码。

(1) 我们首先声明了几个新的变量：

```
Properties {  
    _MainTex ("Main Tex", 2D) = "white" {}  
    _Color ("Color Tint", Color) = (1, 1, 1, 1)  
    _VerticalBillboarding ("Vertical Restraints", Range(0, 1)) = 1  
}
```

其中，\_MainTex是广告牌显示的透明纹理，\_Color用于控制显示整体颜色，\_VerticalBillboarding则用于调整是固定法线还是固定指向上的方向，即约束垂直方向的程度。

(2) 在本例中，我们需要为透明效果设置合适的SubShader标签：

```
SubShader {  
    // Need to disable batching because of the vertex animation  
    Tags {"Queue"="Transparent" "IgnoreProjector"="True"  
"RenderType"="Transparent" "DisableBatching"="True"}
```

在上面的设置中，我们除了为透明效果设置Queue、IgnoreProjector和RenderType外，还设置了一个新的标签——**DisableBatching**。我们在3.3.3节中介绍过该标签的含义：一些SubShader在使用Unity的批处理功能时会出现问题，这时可以通过该标签来直接指明是否对该SubShader使用批处理。而这些需要特殊处理的Shader通常就是指包含了模型空间的顶点动画的Shader。这是因为，批处理会合并所有相关的模型，而这些模型各自的模型空间就会被丢失。而在广告牌技术中，我们需要使用物体的模型空间下的位置来作为锚点进行计算。因此。在这里需要取消对该Shader的批处理操作。

(3) 接着，我们设置了Pass的渲染状态：

```
Pass {  
    Tags { "LightMode"="ForwardBase" }  
  
    ZWrite Off  
    Blend SrcAlpha OneMinusSrcAlpha  
    Cull Off
```

这里关闭了深度写入，开启并设置了混合模式，并关闭了剔除功能。这是为了让广告牌的每个面都能显示。

(4) 顶点着色器是我们的核心，所有的计算都是在模型空间下进行的。我们首先选择模型空间的原点作为广告牌的锚点，并利用内置变量获取模型空间下的视角位置：

```
// Suppose the center in object space is fixed  
float3 center = float3(0, 0, 0);
```

```
float3 viewer = mul(_World2Object,float4(_WorldSpaceCameraPos, 1));
```

然后，我们开始计算3个正交矢量。首先，我们根据观察位置和锚点计算目标法线方向，并根据\_VerticalBillboarding属性来控制垂直方向上的约束度。

```
float3 normalDir = viewer - center;  
// If _VerticalBillboarding equals 1, we use the desired view dir  
// as the normal dir  
// Which means the normal dir is fixed  
// Or if _VerticalBillboarding equals 0, the y of normal is 0  
// Which means the up dir is fixed  
normalDir.y = normalDir.y * _VerticalBillboarding;  
normalDir = normalize(normalDir);
```

当\_VerticalBillboarding为1时，意味着法线方向固定为视角方向；当\_VerticalBillboarding为0时，意味着向上方向固定为(0, 1, 0)。最后，我们需要对计算得到的法线方向进行归一化操作来得到单位矢量。

接着，我们得到了粗略的向上方向。为了防止法线方向和向上方向平行（如果平行，那么叉积得到的结果将是错误的），我们对法线方向的y分量进行判断，以得到合适的向上方向。然后，根据法线方向和粗略的向上方向得到向右方向，并对结果进行归一化。但由于此时向上的方向还是不准确的，我们又根据准确的法线方向和向右方向得到最后的向上方向：

```
// Get the approximate up dir  
// If normal dir is already towards up, then the up dir is towards  
// front  
float3 upDir = abs(normalDir.y) > 0.999 ? float3(0, 0, 1) :  
float3(0, 1, 0);  
float3 rightDir = normalize(cross(upDir, normalDir));  
upDir = normalize(cross(normalDir, rightDir));
```

这样，我们得到了所需的3个正交基矢量。我们根据原始的位置相对于锚点的偏移量以及3个正交基矢量，以计算得到新的顶点位置：

```
float3 centerOffs = v.vertex.xyz - center;
float3 localPos = center + rightDir * centerOffs.x + upDir *
centerOffs.y + normalDir * centerOffs.z;
```

最后，把模型空间的顶点位置变换到裁剪空间中：

```
o.pos = mul(UNITY_MATRIX_MVP, float4(localPos, 1));
```

(5) 片元着色器的代码非常简单，我们只需要对纹理进行采样，再与颜色值相乘即可：

```
fixed4 frag (v2f i) : SV_Target {
    fixed4 c = tex2D (_MainTex, i.uv);
    c.rgb *= _Color.rgb;

    return c;
}
```

(6) 最后，我们把Fallback设置为内置的Transparent/VertexLit（也可以选择关闭Fallback）：

```
Fallback "Transparent/VertexLit"
```

需要说明的是，在上面的例子中，我们使用的是Unity自带的四边形（Quad）来作为广告牌，而不能使用自带的平面（Plane）。这是因为，我们的代码是建立在一个竖直摆放的多边形的基础上的，也就是说，这个多边形的顶点结构需要满足在模型空间下是竖直排列的。只有这样，我们才能使用v.vertex来计算得到正确的相对于中心的位置偏移量。

保存后返回场景，把本书资源中的 `Assets/Textures/Chapter11/star.png` 拖曳到材质的 `Main Tex` 中，即可得到类似图11.6中的效果。

### 11.3.3 注意事项

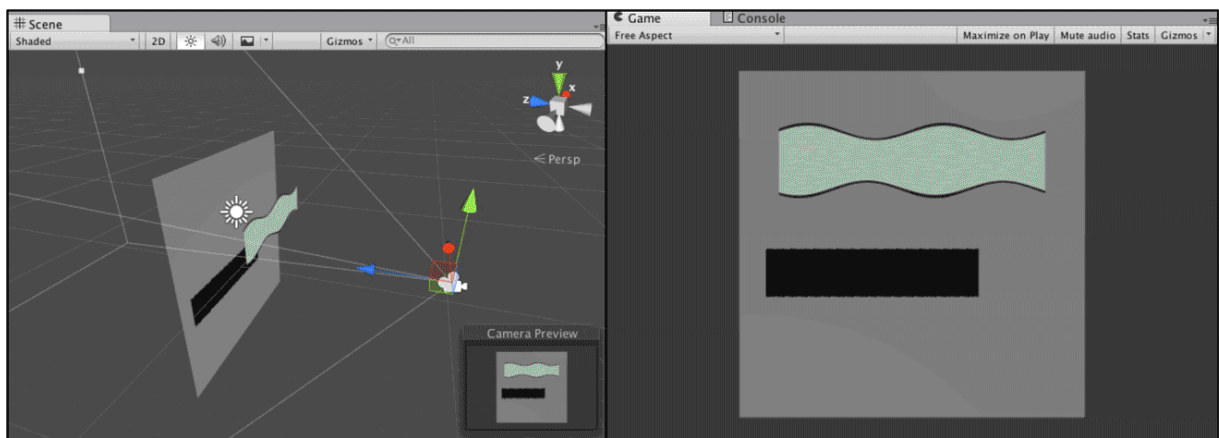
顶点动画虽然非常灵活有效，但有一些注意事项需要在此提醒读者。

首先，如11.3.2节看到的那样，如果我们在模型空间下进行了一些顶点动画，那么批处理往往就会破坏这种动画效果。这时，我们可以通过 `SubShader` 的 `DisableBatching` 标签来强制取消对该 `Unity Shader` 的批处理。然而，取消批处理会带来一定的性能下降，增加了 `Draw Call`，因此我们应该尽量避免使用模型空间下的一些绝对位置和方向来进行计算。在广告牌的例子中，为了避免显式使用模型空间的中心来作为锚点，我们可以利用顶点颜色来存储每个顶点到锚点的距离值，这种做法在商业游戏中很常见。

其次，如果我们想要对包含了顶点动画的物体添加阴影，那么如果仍然像9.4节中那样使用内置的 `Diffuse` 等包含的阴影 `Pass` 来渲染，就得不到正确的阴影效果（这里指的是无法向其他物体正确地投射阴影）。这是因为，我们讲过 `Unity` 的阴影绘制需要调用一个 `ShadowCaster Pass`，而如果直接使用这些内置的 `ShadowCaster Pass`，这个 `Pass` 中并没有进行相关的顶点动画，因此 `Unity` 会仍然按照原来的顶点位置来计算阴影，这并不是我们希望看到的。这时，我们就需要提供一个自定义的 `ShadowCaster Pass`，在这个 `Pass` 中，我们将进行同样的顶点变换过程。需要注意的是，在前面的实现中，如果涉及半透明物

体我们都把Fallback设置成了Transparent/VertexLit，而Transparent/VertexLit没有定义ShadowCaster Pass，因此也就不会产生阴影（详见9.4.5节）。

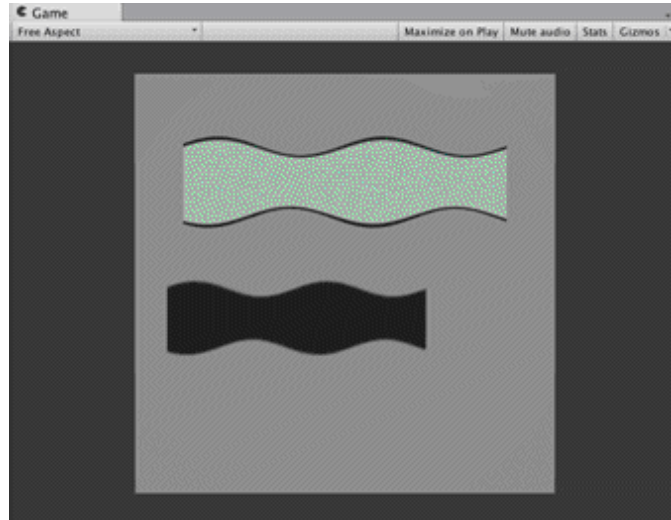
在本书资源的Scene\_11\_3\_3场景中，我们给出了计算顶点动画的阴影的一个例子。在这个例子中，我们使用了11.3.1节中的大部分代码，模拟一个波动的水流。同时，我们开启了场景中平行光的阴影效果，并添加了一个平面来接收来自“水流”的阴影。我们还把这个Unity Shader的Fallback设置为了内置的VertexLit，这样 Unity 将根据Fallback最终找到VertexLit中的ShadowCaster Pass来渲染阴影。图11.7给出了这样的结果。



▲ 图11.7 当进行顶点动画时，如果仍然使用内置的ShadowCaster Pass来渲染阴影，可能会得到错误的阴影效果

可以看出，此时虽然Water模型发生了形变，但它的阴影并没有产生相应的动画效果。为了正确绘制变形对象的阴影，我们就需要提供自定义的ShadowCaster Pass。读者可以在本书资源的Chapter11-

VertexAnimationWithShadow中找到对应的Unity Shader。使用该Shader得到的阴影效果如图11.8所示。



▲ 图11.8 使用自定义的ShadowCaster Pass为变形物体绘制正确的阴影

在这个Shader中，我们提供了一个ShadowCaster Pass，相关代码如下：

```
// Pass to render object as a shadow caster
Pass {
    Tags { "LightMode" = "ShadowCaster" }

    CGPROGRAM

    #pragma vertex vert
    #pragma fragment frag

    #pragma multi_compile_shadowcaster

    #include "UnityCG.cginc"

    float _Magnitude;
    float _Frequency;
    float _InvWaveLength;
    float _Speed;

    struct a2v {
        float4 vertex : POSITION;
```



```

        float4 texcoord : TEXCOORD0;
    };

    struct v2f {
        V2F_SHADOW_CASTER;
    };

    v2f vert(a2v v) {
        v2f o;

        float4 offset;

        offset.yzw = float3(0.0, 0.0, 0.0);

        offset.x = sin(_Frequency * _Time.y + v.vertex.x *
_InvWaveLength + v.vertex.y
        * _InvWaveLength + v.vertex.z * _InvWaveLength) *
_Magnitude;

        v.vertex = v.vertex + offset;

        TRANSFER_SHADOW_CASTER_NORMALOFFSET(o)

        return o;
    }

    fixed4 frag(v2f i) : SV_Target {
        SHADOW_CASTER_FRAGMENT(i)
    }
    ENDCG
}

```

阴影投射的重点在于我们需要按正常Pass的处理来剔除片元或进行顶点动画，以便阴影可以和物体正常渲染的结果相匹配。在自定义的阴影投射的Pass中，我们通常会使用Unity提供的内置宏V2F\_SHADOW\_CASTER、TRANSFER\_SHADOW\_CASTER\_NORMALOFFSET（旧版本中会使用TRANSFER\_SHADOW\_CASTER）和SHADOW\_CASTER\_FRAGMENT来计算阴影投射时需要的各种变量，而我们可以只关注自定义计算的部分。在上面的代码中，我们首先在v2f结构体中利用V2F\_SHADOW\_CASTER来定义阴影投射需要定义的

变量。随后，在顶点着色器中，我们首先按之前对顶点的处理方法计算顶点的偏移量，不同的是，我们直接把偏移值加到顶点位置变量中，再使用TRANSFER\_SHADOW\_CASTER\_NORMALOFFSET来让Unity为我们完成剩下的事情。在片元着色器中，我们直接使用SHADOW\_CASTER\_FRAGMENT来让Unity自动完成阴影投射的部分，把结果输出到深度图和阴影映射纹理中。

通过Unity提供的这3个内置宏（在UnityCG.cginc文件中被定义），我们可以方便地自定义需要的阴影投射的Pass，但由于这些宏里需要使用一些特定的输入变量，因此我们需要保证为它们提供了这些变量。例如，TRANSFER\_SHADOW\_CASTER\_NORMALOFFSET会使用名称v作为输入结构体，v中需要包含顶点位置v.vertex和顶点法线v.normal的信息，我们可以直接使用内置的appdata\_base结构体，它包含了这些必需的顶点变量。如果我们需要进行顶点动画，可以在顶点着色器中直接修改v.vertex，再传递给TRANSFER\_SHADOW\_CASTER\_NORMALOFFSET即可。在15.1节中，我们还会看到如何在阴影投射的Pass中剔除片元，以实现自定义的透明度测试效果。

## 第4篇 高级篇

高级篇涵盖了一些Shader的高级用法，例如，如何实现屏幕特效、利用法线和深度缓冲，以及非真实感渲染等，同时，我们还会介绍一些针对移动平台的优化技巧。

### 第12章 屏幕后处理效果

这一章将介绍如何在Unity中实现一个基本的屏幕后处理脚本系统，并给出一些基本的屏幕特效的实现原理，如高斯模糊、边缘检测等。

### 第13章 使用深度和法线纹理

本章将介绍如何在Unity中获取这些特殊的纹理来实现屏幕特效。

### 第14章 非真实感渲染

这一章将会给出常见的非真实感渲染的算法，如卡通渲染、素描风格的渲染等。

### 第15章 使用噪声

很多时候噪声是我们实现特效的“救星”。本章给出了噪声在游戏渲染中的一些应用。

### 第16章 Unity中的渲染优化技术

优化往往是游戏渲染中的重点。这一章介绍了Unity中针对移动平台常见的优化技巧。

## 第12章 屏幕后处理效果

**屏幕后处理效果**（**screen post-processing effects**）是游戏中实现屏幕特效的常见方法。在本章中，我们将学习如何在Unity中利用渲染纹理来实现各种常见的屏幕后处理效果。在12.1节中，我们首先会解释在Unity中实现屏幕后处理效果的原理，并建立一个基本的屏幕后处理脚本系统。随后在12.2节中，我们会使用这个系统实现一个简单的调整画面亮度、饱和度和对比度的屏幕特效。在12.3节中，我们会接触到图像滤波的概念，并利用 Sobel算子在屏幕空间中对图像进行边缘检测，实现描边效果。在此基础上，12.4节将会介绍如何实现一个高斯模糊的屏幕特效。在12.5和12.6节中，我们会分别介绍如何实现Bloom和运动模糊效果。

### 12.1 建立一个基本的屏幕后处理脚本系统

屏幕后处理，顾名思义，通常指的是在渲染完整个场景得到屏幕图像后，再对这个图像进行一系列操作，实现各种屏幕特效。使用这种技术，可以为游戏画面添加更多的艺术效果，例如景深（Depth of Field）、运动模糊（Motion Blur）等。

因此，想要实现屏幕后处理的基础在于得到渲染后的屏幕图像，即抓取屏幕，而Unity为我们提供了这样一个方便的接口——**OnRenderImage函数**。它的函数声明如下：

```
MonoBehaviour.OnRenderImage (RenderTexture src, RenderTexture dest)
```

当我们在脚本中声明此函数后，Unity会把当前渲染得到的图像存储在第一个参数对应的源渲染纹理中，通过函数中的一系列操作后，再把目标渲染纹理，即第二个参数对应的渲染纹理显示到屏幕上。在OnRenderImage函数中，我们通常是利用**Graphics.Blit函数**来完成对渲染纹理的处理。它有3种函数声明：

```
public static void Blit(Texture src, RenderTexture dest);  
public static void Blit(Texture src, RenderTexture dest, Material  
mat, int pass = -1);  
public static void Blit(Texture src, Material mat, int pass = -1);
```

其中，参数src对应了源纹理，在屏幕后处理技术中，这个参数通常就是当前屏幕的渲染纹理或是上一步处理后得到的渲染纹理。参数dest是目标渲染纹理，如果它的值为null就会直接将结果显示在屏幕上。参数mat是我们使用的材质，这个材质使用的Unity Shader将会进行各种屏幕后处理操作，而src纹理将会被传递给Shader中名为\_MainTex的纹理属性。参数pass的默认值为-1，表示将会依次调用Shader内的所有Pass。否则，只会调用给定索引的Pass。

在默认情况下，OnRenderImage函数会在所有的不透明和透明的Pass执行完毕后被调用，以便对场景中所有游戏对象都产生影响。但有时，我们希望在透明的Pass（即渲染队列小于等于2500的Pass，内置的Background、Geometry和AlphaTest渲染队列均在此范围内）执行完毕后立即调用OnRenderImage函数，从而不对透明物体产生任何影响。此时，我们可以在OnRenderImage函数前添加ImageEffectOpaque属性来实现这样的目的。13.4节展示了这样一个例子，在13.4节中，我们会利用深度和法线纹理进行边缘检测从而实现描边的效果，但我们不希望透明物体也被描边。

因此，要在Unity中实现屏幕后处理效果，**过程通常如下**：我们首先需要在摄像中添加一个用于屏幕后处理的脚本。在这个脚本中，我们会实现OnRenderImage函数来获取当前屏幕的渲染纹理。然后，再调用Graphics.Blit函数使用特定的Unity Shader来对当前图像进行处理，再把返回的渲染纹理显示到屏幕上。对于一些复杂的屏幕特效，我们可能需要多次调用Graphics.Blit函数来对上一步的输出结果进行下一步处理。

但是，在进行屏幕后处理之前，我们需要检查一系列条件是否满足，例如当前平台是否支持渲染纹理和屏幕特效，是否支持当前使用的Unity Shader等。为此，我们创建了一个用于屏幕后处理效果的基类，在实现各种屏幕特效时，我们只需要继承自该基类，再实现派生类中不同的操作即可。读者可在本书资源的Assets/Scripts/Chapter12/PostEffectsBase.cs中找到该脚本。

PostEffectsBase.cs的主要代码如下。

(1) 首先，所有屏幕后处理效果都需要绑定在某个摄像机上，并且我们希望在编辑器状态下也可以执行该脚本来查看效果：

```
[ExecuteInEditMode]
[RequireComponent (typeof(Camera))]
public class PostEffectsBase : MonoBehaviour {
```

(2) 为了提前检查各种资源和条件是否满足，我们在Start函数中调用CheckResources函数：

```
// Called when start
protected void CheckResources() {
    bool isSupported = CheckSupport();
```



```

        if (isSupported == false) {
            NotSupported();
        }
    }

    // Called in CheckResources to check support on this platform
    protected bool CheckSupport() {
        if (SystemInfo.supportsImageEffects == false ||
            SystemInfo.supportsRenderTextures == false) {
            Debug.LogWarning("This platform does not support image
effects or render textures.");
            return false;
        }

        return true;
    }

    // Called when the platform doesn't support this effect
    protected void NotSupported() {
        enabled = false;
    }

    protected void Start() {
        CheckResources();
    }
}

```

一些屏幕特效可能需要更多的设置，例如设置一些默认值等，可以重载Start、CheckResources或CheckSupport函数。

(3) 由于每个屏幕后处理效果通常都需要指定一个Shader来创建一个用于处理渲染纹理的材质，因此基类中也提供了这样的方法：

```

// Called when need to create the material used by this effect
protected Material CheckShaderAndCreateMaterial(Shader shader,
Material material) {
    if (shader == null) {
        return null;
    }

    if (shader.isSupported && material && material.shader ==
shader)
        return material;

    if (!shader.isSupported) {
        return null;
    }
}

```

```
    else {  
        material = new Material(shader);  
        material.hideFlags = HideFlags.DontSave;  
        if (material)  
            return material;  
        else  
            return null;  
    }  
}
```

`CheckShaderAndCreateMaterial`函数接受两个参数，第一个参数指定了该特效需要使用的`Shader`，第二个参数则是用于后期处理的材质。该函数首先检查`Shader`的可用性，检查通过后就返回一个使用了该`Shader`的材质，否则返回`null`。

在12.2节中，我们就会看到如何继承`PostEffectsBase.cs`来创建一个简单的用于调整屏幕的亮度、饱和度和对比度的特效脚本。

## 12.2 调整屏幕的亮度、饱和度和对比度

在12.1节中，我们了解了实现屏幕后处理特效的技术原理。在本节中，我们就小试牛刀来实现一个非常简单的屏幕特效——调整屏幕的亮度、饱和度和对比度。在本节结束后，我们将得到类似图12.1中的效果。



▲图12.1 左图：原效果。右图：调整了亮度（值为1.2）、饱和度（值为1.6）和对比度（值为1.2）后的效果

为此，我们需要进行如下准备工作。

（1）新建一个场景。在本书资源中，该场景名为Scene\_12\_2。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window → Lighting → Skybox中去掉场景中的天空盒子。

（2）把本书资源中的Assets/Textures/Chapter12/Sakura0.jpg拖曳到场景中，并调整其的位置使它可以填充整个场景。注意，Sakura0.jpg的纹理类型已被设置为Sprite，因此可以直接拖曳到场景中。

（3）新建一个脚本。在本书资源中，该脚本名为BrightnessSaturationAndContrast.cs。把该脚本拖曳到摄像机上。

（4）新建一个Unity Shader。在本书资源中，该Shader名为Chapter12-BrightnessSaturationAndContrast。

我们首先来编写BrightnessSaturationAndContrast.cs脚本。打开该脚本，并进行如下修改。

(1) 首先，继承12.1节中创建的基类：

```
public class BrightnessSaturationAndContrast : PostEffectsBase {
```

(2) 声明该效果需要的Shader，并据此创建相应的材质：

```
public Shader briSatConShader;
private Material briSatConMaterial;
public Material material {
    get {
        briSatConMaterial =
        CheckShaderAndCreateMaterial(briSatConShader, briSatConMaterial);
        return briSatConMaterial;
    }
}
```

在上述代码中，briSatConShader是我们指定的Shader，对应了后面将会实现的Chapter12- BrightnessSaturationAndContrast。

briSatConMaterial是创建的材质，我们提供了名为material的材质来访问它，material的get函数调用了基类的CheckShaderAndCreateMaterial函数来得到对应的材质。

(3) 我们还在脚本中提供了调整亮度、饱和度和对比度的参数：

```
[Range(0.0f, 3.0f)]
public float brightness = 1.0f;

[Range(0.0f, 3.0f)]
public float saturation = 1.0f;

[Range(0.0f, 3.0f)]
public float contrast = 1.0f;
```

我们利用Unity提供的Range属性为每个参数提供了合适的变化区间。

(4) 最后，我们定义OnRenderImage函数来进行真正的特效处理：

```
void OnRenderImage(RenderTexture src, RenderTexture dest) {  
    if (material != null) {  
        material.SetFloat("_Brightness", brightness);  
        material.SetFloat("_Saturation", saturation);  
        material.SetFloat("_Contrast", contrast);  
  
        Graphics.Blit(src, dest, material);  
    } else {  
        Graphics.Blit(src, dest);  
    }  
}
```

每当OnRenderImage函数被调用时，它会检查材质是否可用。如果可用，就把参数传递给材质，再调用Graphics.Blit进行处理；否则，直接把原图像显示到屏幕上，不做任何处理。

下面，我们来实现Shader的部分。打开Chapter12-BrightnessSaturationAndContrast，进行如下修改。

(1) 我们首先需要声明本例使用的各个属性：

```
Properties {  
    _MainTex ("Base (RGB)", 2D) = "white" {}  
    _Brightness ("Brightness", Float) = 1  
    _Saturation ("Saturation", Float) = 1  
    _Contrast ("Contrast", Float) = 1  
}
```

在12.1节中，我们提到Graphics.Blit(src, dest, material)将把第一个参数传递给Shader中名为\_MainTex的属性。因此，我们必须声明一个名为

`_MainTex`的纹理属性。除此之外，我们还声明了用于调整亮度、饱和度和对比度的属性。这些值将会由脚本传递而得。事实上，我们可以省略**Properties**中的属性声明，**Properties**中声明的属性仅仅是为了显示在材质面板中，但对于屏幕特效来说，它们使用的材质都是临时创建的，我们也不需要材质面板上调整参数，而是直接从脚本传递给**Unity Shader**。

(2) 定义用于屏幕后处理的**Pass**:

```
SubShader {  
    Pass {  
        ZTest Always Cull Off ZWrite Off
```

屏幕后处理实际上是在场景中绘制了一个与屏幕同宽同高的四边形面片，为了防止它对其他物体产生影响，我们需要设置相关的渲染状态。在这里，我们关闭了深度写入，是为了防止它“挡住”在其后面被渲染的物体。例如，如果当前的**OnRenderImage**函数在所有不透明的**Pass**执行完毕后立即被调用，不关闭深度写入就会影响后面透明的**Pass**的渲染。这些状态设置可以认为是用于屏幕后处理的**Shader**的“标配”。

(3) 为了在代码中访问各个属性，我们需要在**CG**代码块中声明对应的变量:

```
sampler2D _MainTex;  
half _Brightness;  
half _Saturation;  
half _Contrast;
```

(4) 定义顶点着色器。屏幕特效使用的顶点着色器代码通常都比较简单，我们只需要进行必需的顶点变换，更重要的是，我们需要把

正确的纹理坐标传递给片元着色器，以便对屏幕图像进行正确的采样：

```
struct v2f {
    float4 pos : SV_POSITION;
    half2 uv: TEXCOORD0;
};

v2f vert(appdata_img v) {
    v2f o;

    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.uv = v.texcoord;

    return o;
}
```

在上面的顶点着色器中，我们使用了Unity内置的appdata\_img结构体作为顶点着色器的输入，读者可以在UnityCG.cginc中找到该结构体的声明，它只包含了图像处理时必需的顶点坐标和纹理坐标等变量。

(5) 接着，我们实现了用于调整亮度、饱和度和对比度的片元着色器：

```
fixed4 frag(v2f i) : SV_Target {
    fixed4 renderTex = tex2D(_MainTex, i.uv);

    // Apply brightness
    fixed3 finalColor = renderTex.rgb * _Brightness;

    // Apply saturation
    fixed luminance = 0.2125 * renderTex.r + 0.7154 * renderTex.g +
0.0721 * renderTex.b;
    fixed3 luminanceColor = fixed3(luminance, luminance,
luminance);
    finalColor = lerp(luminanceColor, finalColor, _Saturation);

    // Apply contrast
    fixed3 avgColor = fixed3(0.5, 0.5, 0.5);
    finalColor = lerp(avgColor, finalColor, _Contrast);
}
```



```
    return fixed4(finalColor, renderTex.a);  
}
```

首先，我们得到对原屏幕图像（存储在\_MainTex中）的采样结果renderTex。然后，利用\_Brightness属性来调整亮度。亮度的调整非常简单，我们只需要把原颜色乘以亮度系数\_Brightness即可。然后，我们计算该像素对应的亮度值（luminance），这是通过对每个颜色分量乘以一个特定的系数再相加得到的。我们使用该亮度值创建了一个饱和度为0的颜色值，并使用\_Saturation属性在其和上一步得到的颜色之间进行插值，从而得到希望的饱和度颜色。对比度的处理类似，我们首先创建一个对比度为0的颜色值（各分量均为0.5），再使用\_Contrast属性在其和上一步得到的颜色之间进行插值，从而得到最终的处理结果。

（6）最后，我们关闭该Unity Shader的Fallback:

```
Fallback Off
```

完成后返回编辑器，并把Chapter12-BrightnessSaturationAndContrast拖曳到摄像机的BrightnessSaturationAndContrast.cs脚本中的briSatConShader参数中。调整各个参数后，我们就可以得到类似图12.1中的效果。

在上面的实现中，我们需要手动把Shader拖曳到脚本的参数上。为了在以后的使用中，当把脚本拖曳到摄像机上时直接使用对应的Shader，我们可以在脚本的面板中设置Shader参数的默认值，如图12.2所示。



▲ 图12.2 为脚本设置Shader的默认值

## 12.3 边缘检测

在12.2节中，我们已经学习了如何实现一个简单的屏幕后处理效果。在本节中，我们会学习一个常见的屏幕后处理效果——边缘检测。边缘检测是描边效果的一种实现方法，在本节结束后，我们可以得到类似图12.3中的效果。



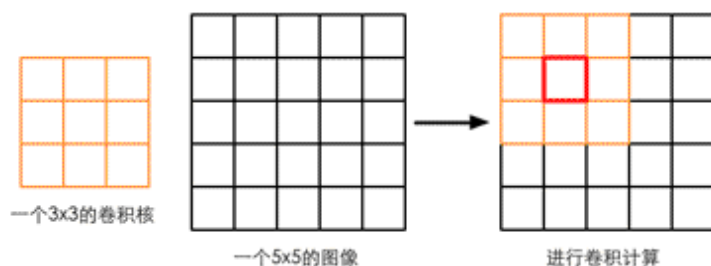
▲ 图12.3 左图：12.2节得到的结果，右图：进行边缘检测后的效果

边缘检测的原理是利用一些边缘检测算子对图像进行**卷积**（**convolution**）操作，我们首先来了解什么是卷积。

### 12.3.1 什么是卷积

在图像处理中，卷积操作指的就是使用一个**卷积核**（**kernel**）对一张图像中的每个像素进行一系列操作。卷积核通常是一个四方形网格结构（例如 $2 \times 2$ 、 $3 \times 3$ 的方形区域），该区域内每个方格都有一个权重

值。当对图像中的某个像素进行卷积时，我们会把卷积核的中心放置于该像素上，如图12.4所示，翻转核之后再依次计算核中每个元素和其覆盖的图像像素值的乘积并求和，得到的结果就是该位置的新像素值。



▲图12.4 卷积核与卷积。使用一个3×3大小的卷积核对一张5×5大小的图像进行卷积操作，当计算图中红色方块对应的像素的卷积结果时，我们首先把卷积核的中心放置在该像素位置，翻转核之后再依次计算核中每个元素和其覆盖的图像像素值的乘积并求和，得到新的像素值

这样的计算过程虽然简单，但可以实现很多常见的图像处理效果，例如图像模糊、边缘检测等。例如，如果我们想要对图像进行均值模糊，可以使用一个3×3的卷积核，核内每个元素的值均为1/9。

### 12.3.2 常见的边缘检测算子

卷积操作的神奇之处在于选择的卷积核。那么，用于边缘检测的卷积核（也被称为边缘检测算子）应该长什么样呢？在回答这个问题前，我们可以首先回想一下边到底是如何形成的。如果相邻像素之间存在差别明显的颜色、亮度、纹理等属性，我们就会认为它们之间应该有一条边界。这种相邻像素之间的差值可以用**梯度（gradient）**来表示，可以想象得到，边缘处的梯度绝对值会比较大。基于这样的理解，有几种不同的边缘检测算子被先后提出来。

Roberts

-1

0

0

1

0

-1

1

0

G<sub>x</sub>

G<sub>y</sub>

Prewitt

-1

0

1

-1

0

1

-1

0

1

-1

-1

-1

0

0

0

1

1

1

G<sub>x</sub>

G<sub>y</sub>

Sobel

-1

0

1

-2

0

2

-1

0

1

-1

-2

-1

0

0

0

1

2

1

G<sub>x</sub>

G<sub>y</sub>

图12.5 3种常见的边缘检测算子

3种常见的边缘检测算子如图12.5所示，它们都包含了两个方向的卷积核，分别用于检测水平方向和竖直方向上的边缘信息。在进行边缘检测时，我们需要对每个像素分别进行一次卷积计算，得到两个方向上的梯度值 $G_x$ 和 $G_y$ ，而整体的梯度可按下面的公式计算而得：

$$G = \sqrt{G_x^2 + G_y^2}$$

由于上述计算包含了开根号操作，出于性能的考虑，我们有时会使用绝对值操作来代替开根号操作：

$$G = |G_x| + |G_y|$$

当得到梯度 $G$ 后，我们就可以据此来判断哪些像素对应了边缘（梯度值越大，越有可能是边缘点）。

### 12.3.3 实现

本节将会使用Sobel算子进行边缘检测，实现描边效果。为此，我们需要进行如下准备工作。

(1) 新建一个场景。在本书资源中，该场景名为Scene\_12\_3。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window -> Lighting -> Skybox中去掉场景中的天空盒子。

(2) 把本书资源中的Assets/Textures/Chapter12/Sakura0.jpg拖曳到场景中，并调整它的位置使其可以填充整个场景。注意，Sakura0.jpg的纹理类型已被设置为Sprite，因此可以直接拖曳到场景中。

(3) 新建一个脚本。在本书资源中，该脚本名为EdgeDetection.cs。把该脚本拖曳到摄像机上。

(4) 新建一个Unity Shader。在本书资源中，该Shader名为Chapter12-EdgeDetection。

我们首先来编写EdgeDetection.cs脚本。打开该脚本，并进行如下修改。

(1) 首先，继承12.1节中创建的基类：

```
public class EdgeDetection : PostEffectsBase {
```

(2) 声明该效果需要的Shader，并据此创建相应的材质：

```
public Shader edgeDetectShader;
private Material edgeDetectMaterial = null;
public Material material {
    get {
        edgeDetectMaterial =
        CheckShaderAndCreateMaterial(edgeDetectShader, edgeDetectMaterial);
        return edgeDetectMaterial;
    }
}
```

在上述代码中，`edgeDetectShader`是我们指定的Shader，对应了后面将会实现的Chapter12-EdgeDetection。

(3) 在脚本中提供用于调整边缘线强度、描边颜色以及背景颜色的参数：

```
[Range(0.0f, 1.0f)]
public float edgesOnly = 0.0f;

public Color edgeColor = Color.black;

public Color backgroundColor = Color.white;
```

当`edgesOnly`值为0时，边缘将会叠加在原渲染图像上；当`edgesOnly`值为1时，则会只显示边缘，不显示原渲染图像。其中，背景颜色由`backgroundColor`指定，边缘颜色由`edgeColor`指定。

(4) 最后，我们定义`OnRenderImage`函数来进行真正的特效处理：

```
void OnRenderImage (RenderTexture src, RenderTexture dest) {
    if (material != null) {
        material.SetFloat("_EdgeOnly", edgesOnly);
        material.SetColor("_EdgeColor", edgeColor);
        material.SetColor("_BackgroundColor", backgroundColor);

        Graphics.Blit(src, dest, material);
    } else {
        Graphics.Blit(src, dest);
    }
}
```

每当`OnRenderImage`函数被调用时，它会检查材质是否可用。如果可用，就把参数传递给材质，再调用`Graphics.Blit`进行处理；否则，直接把原图像显示到屏幕上，不做任何处理。

下面，我们来实现Shader的部分。打开Chapter12-EdgeDetection，进行如下修改。

(1) 我们首先需要声明本例使用的各个属性：

```
Properties {  
    _MainTex ("Base (RGB)", 2D) = "white" {}  
    _EdgeOnly ("Edge Only", Float) = 1.0  
    _EdgeColor ("Edge Color", Color) = (0, 0, 0, 1)  
    _BackgroundColor ("Background Color", Color) = (1, 1, 1, 1)  
}
```

\_MainTex对应了输入的渲染纹理。

(2) 定义用于屏幕后处理的Pass，设置相关的渲染状态：

```
SubShader {  
    Pass {  
        ZTest Always Cull Off ZWrite Off
```

(3) 为了在代码中访问各个属性，我们需要在CG代码块中声明对应的变量：

```
sampler2D _MainTex;  
half4 _MainTex_TexelSize;  
fixed _EdgeOnly;  
fixed4 _EdgeColor;  
fixed4 _BackgroundColor;
```

在上面的代码中，我们还声明了一个新的变量 \_MainTex\_TexelSize。xxx\_TexelSize是Unity为我们提供的访问xxx纹理对应的每个纹素的大小。例如，一张512×512大小的纹理，该值大约为0.001953（即1/512）。由于卷积需要对相邻区域内的纹理进行采样，因此我们需要利用\_MainTex\_TexelSize来计算各个相邻区域的纹理坐标。



(4) 在顶点着色器的代码中，我们计算了边缘检测时需要的纹理坐标：

```
struct v2f {
    float4 pos : SV_POSITION;
    half2 uv[9] : TEXCOORD0;
};

v2f vert(appdata_img v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    half2 uv = v.texcoord;

    o.uv[0] = uv + _MainTex_TexelSize.xy * half2(-1, -1);
    o.uv[1] = uv + _MainTex_TexelSize.xy * half2(0, -1);
    o.uv[2] = uv + _MainTex_TexelSize.xy * half2(1, -1);
    o.uv[3] = uv + _MainTex_TexelSize.xy * half2(-1, 0);
    o.uv[4] = uv + _MainTex_TexelSize.xy * half2(0, 0);
    o.uv[5] = uv + _MainTex_TexelSize.xy * half2(1, 0);
    o.uv[6] = uv + _MainTex_TexelSize.xy * half2(-1, 1);
    o.uv[7] = uv + _MainTex_TexelSize.xy * half2(0, 1);
    o.uv[8] = uv + _MainTex_TexelSize.xy * half2(1, 1);

    return o;
}
```

我们在v2f结构体中定义了一个维数为9的纹理数组，对应了使用Sobel算子采样时需要的9个邻域纹理坐标。通过把计算采样纹理坐标的代码从片元着色器中转移到顶点着色器中，可以减少运算，提高性能。由于从顶点着色器到片元着色器的插值是线性的，因此这样的转移并不会影响纹理坐标的计算结果。

(5) 片元着色器是我们的重点，它的代码如下：

```
fixed4 fragSobel(v2f i) : SV_Target {
    half edge = Sobel(i);

    fixed4 withEdgeColor = lerp(_EdgeColor, tex2D(_MainTex,
i.uv[4]), edge);
    fixed4 onlyEdgeColor = lerp(_EdgeColor, _BackgroundColor,
```

```
edge);  
    return lerp(withEdgeColor, onlyEdgeColor, _EdgeOnly);  
}
```

我们首先调用Sobel函数计算当前像素的梯度值`edge`，并利用该值分别计算了背景为原图和纯色下的颜色值，然后利用`_EdgeOnly`在两者之间插值得到最终的像素值。Sobel函数将利用Sobel算子对原图进行边缘检测，它的定义如下：

```
fixed luminance(fixed4 color) {  
    return 0.2125 * color.r + 0.7154 * color.g + 0.0721 * color.b;  
}  
  
half Sobel(v2f i) {  
    const half Gx[9] = {-1, -2, -1,  
                        0,  0,  0,  
                        1,  2,  1};  
    const half Gy[9] = {-1,  0,  1,  
                        -2,  0,  2,  
                        -1,  0,  1};  
  
    half texColor;  
    half edgeX = 0;  
    half edgeY = 0;  
    for (int it = 0; it < 9; it++) {  
        texColor = luminance(tex2D(_MainTex, i.uv[it]));  
        edgeX += texColor * Gx[it];  
        edgeY += texColor * Gy[it];  
    }  
  
    half edge = 1 - abs(edgeX) - abs(edgeY);  
  
    return edge;  
}
```

我们首先定义了水平方向和竖直方向使用的卷积核 $G_x$ 和 $G_y$ 。接着，我们依次对9个像素进行采样，计算它们的亮度值，再与卷积核 $G_x$ 和 $G_y$ 中对应的权重相乘后，叠加到各自的梯度值上。最后，我们从1中减去水平方向和竖直方向的梯度值的绝对值，得到`edge`。edge值越小，表明该位置越可能是一个边缘点。至此，边缘检测过程结束。

(6) 当然，我们也关闭了该Shader的Fallback:

Fallback Off

完成后返回编辑器，并把Chapter12-EdgeDetection拖曳到摄像机的EdgeDetection.cs脚本中的edgeDetectShader参数中。当然，我们可以在EdgeDetection.cs的脚本面板中将edgeDetectShader参数的默认值设置为Chapter12-EdgeDetection，这样就不需要以后使用时每次都手动拖曳了。图12.6显示了edgeOnly参数为1时对应的屏幕效果。



▲图12.6 只显示边缘的屏幕效果

需要注意的是，本节实现的边缘检测仅仅利用了屏幕颜色信息，而在实际应用中，物体的纹理、阴影等信息均会影响边缘检测的结果，使得结果包含许多非预期的描边。为了得到更加准确的边缘信息，我们往往会在屏幕的深度纹理和法线纹理上进行边缘检测。我们将会在13.4节中实现这种方法。

## 12.4 高斯模糊

在12.3节中，我们学习了卷积的概念，并利用卷积实现了一个简单的边缘检测效果。在本节中，我们将学习卷积的另一个常见应用——高斯模糊。模糊的实现有很多方法，例如均值模糊和中值模糊。均值模糊同样使用了卷积操作，它使用的卷积核中的各个元素值都相等，且相加等于1，也就是说，卷积后得到的像素值是其邻域内各个像素值的平均值。而中值模糊则是选择邻域内对所有像素排序后的中值替换掉原颜色。一个更高级的模糊方法是高斯模糊。在学习完本节后，我们可以得到类似图12.7中的效果。



▲ 图12.7 左边为原效果，右边为高斯模糊后的效果

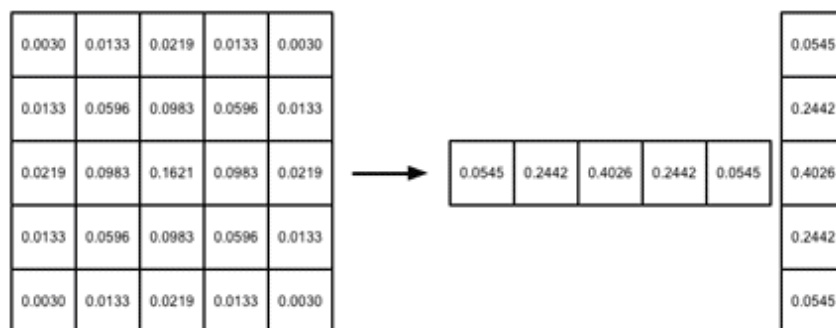
### 12.4.1 高斯滤波

高斯模糊同样利用了卷积计算，它使用的卷积核名为高斯核。高斯核是一个正方形大小的滤波核，其中每个元素的计算都是基于下面的高斯方程：

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

其中， $\sigma$ 是标准方差（一般取值为1）， $x$ 和 $y$ 分别对应了当前位置到卷积核中心的整数距离。要构建一个高斯核，我们只需要计算高斯核中各个位置对应的高斯值。为了保证滤波后的图像不会变暗，我们需要对高斯核中的权重进行归一化，即让每个权重除以所有权重的和，这样可以保证所有权重的和为1。因此，高斯函数中 $e$ 前面的系数实际不会对结果有任何影响。图12.8显示了一个标准方差为1的5×5大小的高斯核。

高斯方程很好地模拟了邻域每个像素对当前处理像素的影响程度——距离越近，影响越大。高斯核的维数越高，模糊程度越大。使用一个 $N \times N$ 的高斯核对图像进行卷积滤波，就需要 $N \times N \times W \times H$ （ $W$ 和 $H$ 分别是图像的宽和高）次纹理采样。当 $N$ 的大小不断增加时，采样次数会变得非常巨大。幸运的是，我们可以把这个二维高斯函数拆分成两个一维函数。也就是说，我们可以使用两个一维的高斯核（图12.8中的右图）先后对图像进行滤波，它们得到的结果和直接使用二维高斯核是一样的，但采样次数只需要 $2 \times N \times W \times H$ 。我们可以进一步观察到，两个一维高斯核中包含了很多重复的权重。对于一个大小为5的一维高斯核，我们实际只需要记录3个权重值即可。



▲图12.8 一个5×5大小的高斯核。左图显示了标准方差为1的高斯核的权重分布，我们可以把这个二维高斯核拆分成两个一维的高斯核（右图）

在本节，我们将会使用上述5×5的高斯核对原图像进行高斯模糊。我们将先后调用两个**Pass**，第一个**Pass**将会使用竖直方向的一维高斯核对图像进行滤波，第二个**Pass**再使用水平方向的一维高斯核对图像进行滤波，得到最终的目标图像。在实现中，我们还将利用图像缩放来进一步提高性能，并通过调整高斯滤波的应用次数来控制模糊程度（次数越多，图像越模糊）。

## 12.4.2 实现

为此，我们需要进行如下准备工作。

（1）新建一个场景。在本书资源中，该场景名为**Scene\_12\_4**。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在**Window → Lighting → Skybox**中去掉场景中的天空盒子。

（2）把本书资源中的**Assets/Textures/Chapter12/Sakura1.jpg**拖曳到场景中，并调整的位置使其可以填充整个场景。注意，**Sakura1.jpg**的纹理类型已被设置为**Sprite**，因此可以直接拖曳到场景中。

（3）新建一个脚本。在本书资源中，该脚本名为**GaussianBlur.cs**。把该脚本拖曳到摄像机上。

（4）新建一个Unity Shader。在本书资源中，该Shader名为**Chapter12-GaussianBlur**。

我们首先来编写GaussianBlur.cs脚本。打开该脚本，并进行如下修改。

(1) 首先，继承12.1节中创建的基类：

```
public class GaussianBlur : PostEffectsBase {
```

(2) 声明该效果需要的Shader，并据此创建相应的材质：

```
public Shader gaussianBlurShader;
private Material gaussianBlurMaterial = null;

public Material material {
    get {
        gaussianBlurMaterial =
        CheckShaderAndCreateMaterial(gaussianBlurShader,
        gaussianBlurMaterial);
        return gaussianBlurMaterial;
    }
}
```

在上述代码中，gaussianBlurShader是我们指定的shader，对应了后面将会实现的Chapter12-GaussianBlur。

(3) 在脚本中，我们还提供了调整高斯模糊迭代次数、模糊范围和缩放系数的参数：

```
// Blur iterations - larger number means more blur.
[Range(0, 4)]
public int iterations = 3;

// Blur spread for each iteration - larger value means more blur
[Range(0.2f, 3.0f)]
public float blurSpread = 0.6f;

[Range(1, 8)]
public int downSample = 2;
```



`blurSpread`和`downSample`都是出于性能的考虑。在高斯核维数不变的情况下，`_BlurSize`越大，模糊程度越高，但采样数却不会受到影响。但过大的`_BlurSize`值会造成虚影，这可能并不是我们希望的。而`downSample`越大，需要处理的像素数越少，同时也能进一步提高模糊程度，但过大的`downSample`可能会使图像像素化。

(4) 最后，我们需要定义关键的`OnRenderImage`函数。我们首先来看第一个版本，也就是最简单的`OnRenderImage`的实现：

```
/// 1st edition: just apply blur
void OnRenderImage(RenderTexture src, RenderTexture dest) {
    if (material != null) {
        int rtW = src.width;
        int rtH = src.height;
        RenderTexture buffer = RenderTexture.GetTemporary(rtW, rtH,
0);

        // Render the vertical pass
        Graphics.Blit(src, buffer, material, 0);
        // Render the horizontal pass
        Graphics.Blit(buffer, dest, material, 1);

        RenderTexture.ReleaseTemporary(buffer);
    } else {
        Graphics.Blit(src, dest);
    }
}
```

与上两节的实现不同，我们这里利用`RenderTexture.GetTemporary`函数分配了一块与屏幕图像大小相同的缓冲区。这是因为，高斯模糊需要调用两个`Pass`，我们需要使用一块中间缓存来存储第一个`Pass`执行完毕后得到的模糊结果。如代码所示，我们首先调用`Graphics.Blit(src, buffer, material, 0)`，使用`Shader`中的第一个`Pass`（即使用竖直方向的一维高斯核进行滤波）对`src`进行处理，并将结果存储在了`buffer`中。然后，再调用`Graphics.Blit(buffer, dest, material, 1)`，使用`Shader`中的第二

个**Pass**（即使用水平方向的一维高斯核进行滤波）对**buffer**进行处理，返回最终的屏幕图像。最后，我们还需要调用**RenderTarget.ReleaseTemporary**来释放之前分配的缓存。

（5）在理解了上述代码后，我们可以实现第二个版本的**OnRenderImage**函数。在这个版本中，我们将利用缩放对图像进行降采样，从而减少需要处理的像素个数，提高性能。

```
/// 2nd edition: scale the render texture
void OnRenderImage (RenderTarget src, RenderTexture dest) {
    if (material != null) {
        int rtW = src.width/downSample;
        int rtH = src.height/downSample;
        RenderTexture buffer = RenderTexture.GetTemporary(rtW, rtH,
0);
        buffer.filterMode = FilterMode.Bilinear;

        // Render the vertical pass
        Graphics.Blit(src, buffer, material, 0);
        // Render the horizontal pass
        Graphics.Blit(buffer, dest, material, 1);

        RenderTexture.ReleaseTemporary(buffer);
    } else {
        Graphics.Blit(src, dest);
    }
}
```

与第一个版本代码不同的是，我们在声明缓冲区的大小时，使用了小于原屏幕分辨率的尺寸，并将该临时渲染纹理的滤波模式设置为双线性。这样，在调用第一个**Pass**时，我们需要处理的像素个数就是原来的几分之一。对图像进行降采样不仅可以减少需要处理的像素个数，提高性能，而且适当的降采样往往还可以得到更好的模糊效果。尽管**downSample**值越大，性能越好，但过大的**downSample**可能会造成图像像素化。

(6) 最后一个版本的代码还考虑了高斯模糊的迭代次数:

```
/// 3rd edition: use iterations for larger blur
void OnRenderImage (RenderTexture src, RenderTexture dest) {
    if (material != null) {
        int rtW = src.width/downSample;
        int rtH = src.height/downSample;

        RenderTexture buffer0 = RenderTexture.GetTemporary(rtW,
rtH, 0);
        buffer0.filterMode = FilterMode.Bilinear;

        Graphics.Blit(src, buffer0);

        for (int i = 0; i < iterations; i++) {
            material.SetFloat("_BlurSize", 1.0f + i * blurSpread);

            RenderTexture buffer1 = RenderTexture.GetTemporary(rtW,
rtH, 0);

            // Render the vertical pass
            Graphics.Blit(buffer0, buffer1, material, 0);

            RenderTexture.ReleaseTemporary(buffer0);
            buffer0 = buffer1;
            buffer1 = RenderTexture.GetTemporary(rtW, rtH, 0);

            // Render the horizontal pass
            Graphics.Blit(buffer0, buffer1, material, 1);

            RenderTexture.ReleaseTemporary(buffer0);
            buffer0 = buffer1;
        }

        Graphics.Blit(buffer0, dest);
        RenderTexture.ReleaseTemporary(buffer0);
    } else {
        Graphics.Blit(src, dest);
    }
}
```

上面的代码显示了如何利用两个临时缓存在迭代之间进行交替的过程。在迭代开始前，我们首先定义了第一个缓存buffer0，并把src中的图像缩放后存储到buffer0中。在迭代过程中，我们又定义了第二个缓存buffer1。在执行第一个Pass时，输入是buffer0，输出是buffer1，完

毕后首先把buffer0释放，再把结果值buffer1存储到buffer0中，重新分配buffer1，然后再调用第二个Pass，重复上述过程。迭代完成后，buffer0将存储最终的图像，我们再利用Graphics.Blit(buffer0, dest)把结果显示到屏幕上，并释放缓存。

下面，我们来实现Shader的部分。打开Chapter12-GaussianBlur，进行如下修改。

(1) 我们首先需要声明本例使用的各个属性：

```
Properties {  
    _MainTex ("Base (RGB)", 2D) = "white" {}  
    _BlurSize ("Blur Size", Float) = 1.0  
}
```

\_MainTex对应了输入的渲染纹理。

(2) 在本节中，我们将第一次使用CGINCLUDE来组织代码。我们在SubShader块中利用CGINCLUDE和ENDCG语义来定义一系列代码：

```
SubShader {  
    CGINCLUDE  
    ...  
    ENDCG  
    ...  
}
```

这些代码不需要包含在任何Pass语义块中，在使用时，我们只需要在Pass中直接指定需要使用的顶点着色器和片元着色器函数名即可。CGINCLUDE类似于C++中头文件的功能。由于高斯模糊需要定义两个Pass，但它们使用的片元着色器代码是完全相同的，使用CGINCLUDE可以避免我们编写两个完全一样的frag函数。

(3) 在CG代码块中，定义与属性对应的变量：

```
sampler2D _MainTex;  
half4 _MainTex_TexelSize;  
float _BlurSize;
```

由于要得到相邻像素的纹理坐标，我们这里再一次使用了Unity提供的`_MainTex_TexelSize`变量，以计算相邻像素的纹理坐标偏移量。

(4) 分别定义两个Pass使用的顶点着色器。下面是竖直方向的顶点着色器代码：

```
struct v2f {  
    float4 pos : SV_POSITION;  
    half2 uv[5]: TEXCOORD0;  
};  
  
v2f vertBlurVertical(appdata_img v) {  
    v2f o;  
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);  
  
    half2 uv = v.texcoord;  
  
    o.uv[0] = uv;  
    o.uv[1] = uv + float2(0.0, _MainTex_TexelSize.y * 1.0) *  
_BlurSize;  
    o.uv[2] = uv - float2(0.0, _MainTex_TexelSize.y * 1.0) *  
_BlurSize;  
    o.uv[3] = uv + float2(0.0, _MainTex_TexelSize.y * 2.0) *  
_BlurSize;  
    o.uv[4] = uv - float2(0.0, _MainTex_TexelSize.y * 2.0) *  
_BlurSize;  
  
    return o;  
}
```

在本节中我们会利用 $5 \times 5$ 大小的高斯核对原图像进行高斯模糊，而由12.4.1节可知，一个 $5 \times 5$ 的二维高斯核可以拆分成两个大小为5的一维高斯核，因此我们只需要计算5个纹理坐标即可。为此，我们在`v2f`结构中定义了一个5维的纹理坐标数组。数组的第一个坐标存储了当前的

采样纹理，而剩余的四个坐标则是高斯模糊中对邻域采样时使用的纹理坐标。我们还和属性\_**BlurSize**相乘来控制采样距离。在高斯核维数不变的情况下，\_**BlurSize**越大，模糊程度越高，但采样数却不会受到影响。但过大的\_**BlurSize**值会造成虚影，这可能并不是我们希望的。通过把计算采样纹理坐标的代码从片元着色器中转移到顶点着色器中，可以减少运算，提高性能。由于从顶点着色器到片元着色器的插值是线性的，因此这样的转移并不会影响纹理坐标的计算结果。

水平方向的顶点着色器和上面的代码类似，只是在计算4个纹理坐标时使用了水平方向的纹素大小进行纹理偏移。

(5) 定义两个Pass共用的片元着色器：

```
fixed4 fragBlur(v2f i) : SV_Target {
    float weight[3] = {0.4026, 0.2442, 0.0545};

    fixed3 sum = tex2D(_MainTex, i.uv[0]).rgb * weight[0];

    for (int it = 1; it < 3; it++) {
        sum += tex2D(_MainTex, i.uv[it*2-1]).rgb *
weight[it];
        sum += tex2D(_MainTex, i.uv[it*2]).rgb * weight[it];
    }

    return fixed4(sum, 1.0);
}
```

由12.4.1节可知，一个5×5的二维高斯核可以拆分成两个大小为5的一维高斯核，并且由于它的对称性，我们只需要记录3个高斯权重，也就是代码中的**weight**变量。我们首先声明了各个邻域像素对应的权重**weight**，然后将结果值**sum**初始化为当前的像素值乘以它的权重值。根据对称性，我们进行了两次迭代，每次迭代包含了两次纹理采样，并

把像素值和权重相乘后的结果叠加到sum中。最后，函数返回滤波结果sum。

(6) 然后，我们定义了高斯模糊使用的两个Pass:

```
ZTest Always Cull Off ZWrite Off

Pass {
    NAME "GAUSSIAN_BLUR_VERTICAL"

    CGPROGRAM

    #pragma vertex vertBlurVertical
    #pragma fragment fragBlur

    ENDCG
}

Pass {
    NAME "GAUSSIAN_BLUR_HORIZONTAL"

    CGPROGRAM

    #pragma vertex vertBlurHorizontal
    #pragma fragment fragBlur

    ENDCG
}
```

注意，我们仍然首先设置了渲染状态。和之前实现不同的是，我们为两个Pass使用NAME语义（见3.3.3节）定义了它们的名字。这是因为，高斯模糊是非常常见的图像处理操作，很多屏幕特效都是建立在它的基础上的，例如Bloom效果（见12.5节）。为Pass定义名字，可以在其他Shader中直接通过它们的名字来使用该Pass，而不需要再重复编写代码。

(7) 最后，关闭该Shader的Fallback:

```
Fallback Off
```



完成后返回编辑器，并把Chapter12-GaussianBlur拖曳到摄像机的GaussianBlur.cs脚本中的gaussianBlurShader参数中。当然，我们可以在GaussianBlur.cs的脚本面板中将gaussianBlurShader参数的默认值设置为Chapter12-GaussianBlur，这样就不需要以后使用时每次都手动拖曳了。

## 12.5 Bloom效果

Bloom特效是游戏中常见的一种屏幕效果。这种特效可以模拟真实摄像机的一种图像效果，它让画面中较亮的区域“扩散”到周围的区域中，造成一种朦胧的效果。图12.9给出了动画短片《大象之梦》（英文名：Elephants Dream）中的一个Bloom效果。

本节将会实现一个基本的Bloom特效，在学习完本节后，我们可以得到类似图12.10中的效果。



▲图12.9 动画短片《大象之梦》中的Bloom效果，光线透过门扩散到了周围较暗的区域中



▲图12.10 左边为原效果，右边为Bloom处理后的效果

Bloom的实现原理非常简单：我们首先根据一个阈值提取出图像中的较亮区域，把它们存储在一张渲染纹理中，再利用高斯模糊对这张渲染纹理进行模糊处理，模拟光线扩散的效果，最后再将其和原图像进行混合，得到最终的效果。

为此，我们需要进行如下准备工作。

(1) 新建一个场景。在本书资源中，该场景名为Scene\_12\_5。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window → Lighting → Skybox中去掉场景中的天空盒子。

(2) 把本书资源中的Textures/Chapter12/Sakura1.jpg拖曳到场景中，并调整它的位置使其可以填充整个场景。注意，Sakura1.jpg的纹理类型已被设置为Sprite，因此可以直接拖曳到场景中。

(3) 新建一个脚本。在本书资源中，该脚本名为**Bloom.cs**。把该脚本拖曳到摄像机上。

(4) 新建一个Unity Shader。在本书资源中，该Shader名为**Chapter12-Bloom**。

我们首先来编写**Bloom.cs**脚本。打开该脚本，并进行如下修改。

(1) 首先，继承12.1节中创建的基类：

```
public class Bloom : PostEffectsBase {
```

(2) 声明该效果需要的Shader，并据此创建相应的材质：

```
public Shader bloomShader;
private Material bloomMaterial = null;
public Material material {
    get {
        bloomMaterial = CheckShaderAndCreateMaterial(bloomShader,
        bloomMaterial);
        return bloomMaterial;
    }
}
```

在上述代码中，**bloomShader**是我们指定的Shader，对应了后面将会实现的**Chapter12-Bloom**。

(3) 由于**Bloom**效果是建立在高斯模糊的基础上的，因此脚本中提供的参数和12.4节中的几乎完全一样，我们只增加了一个新的参数**luminanceThreshold**来控制提取较亮区域时使用的阈值大小：

```
// Blur iterations - larger number means more blur.
[Range(0, 4)]
public int iterations = 3;

// Blur spread for each iteration - larger value means more blur
```

```
[Range(0.2f, 3.0f)]
public float blurSpread = 0.6f;

[Range(1, 8)]
public int downSample = 2;

[Range(0.0f, 4.0f)]
public float luminanceThreshold = 0.6f;
```

尽管在绝大多数情况下，图像的亮度值不会超过1。但如果我们开启了HDR，硬件会允许我们把颜色值存储在一个更高精度范围的缓冲中，此时像素的亮度值可能会超过1。因此，在这里我们把luminanceThreshold的值规定在[0, 4]范围内。更多关于HDR的内容，可以参见18.4.3节。

(4) 最后，我们需要定义关键的OnRenderImage函数：

```
void OnRenderImage (RenderTexture src, RenderTexture dest) {
    if (material != null) {
        material.SetFloat("_LuminanceThreshold",
luminanceThreshold);

        int rtW = src.width/downSample;
        int rtH = src.height/downSample;

        RenderTexture buffer0 = RenderTexture.GetTemporary(rtW,
rtH, 0);
        buffer0.filterMode = FilterMode.Bilinear;

        Graphics.Blit(src, buffer0, material, 0);

        for (int i = 0; i < iterations; i++) {
            material.SetFloat("_BlurSize", 1.0f + i * blurSpread);

            RenderTexture buffer1 = RenderTexture.GetTemporary(rtW,
rtH, 0);

            // Render the vertical pass
            Graphics.Blit(buffer0, buffer1, material, 1);

            RenderTexture.ReleaseTemporary(buffer0);
            buffer0 = buffer1;
            buffer1 = RenderTexture.GetTemporary(rtW, rtH, 0);
```

```

        // Render the horizontal pass
        Graphics.Blit(buffer0, buffer1, material, 2);

        RenderTexture.ReleaseTemporary(buffer0);
        buffer0 = buffer1;
    }

    material.SetTexture ("_Bloom", buffer0);
    Graphics.Blit (src, dest, material, 3);

    RenderTexture.ReleaseTemporary(buffer0);
} else {
    Graphics.Blit(src, dest);
}
}

```

上面的代码和12.4节中进行高斯模糊时使用的代码基本相同，但进行了一些修改。我们前面提到，**Bloom**效果需要3个步骤：首先，提取图像中较亮的区域，因此我们没有像12.4节那样直接对src进行降采样，而是通过调用Graphics.Blit(src, buffer0, material, 0)来使用Shader中的第一个Pass提取图像中的较亮区域，提取得到的较亮区域将存储在buffer0中。然后，我们进行和12.4节中完全一样的高斯模糊迭代处理，这些Pass对应了Shader的第二个和第三个Pass。模糊后的较亮区域将会存储在buffer0中，此时，我们再把buffer0传递给材质中的\_Bloom纹理属性，并调用Graphics.Blit (src, dest, material, 3)使用Shader中的第四个Pass来进行最后的混合，将结果存储在目标渲染纹理dest中。最后，释放临时缓存。

下面，我们来实现Shader的部分。打开Chapter12-Bloom，进行如下修改。

(1) 我们首先需要声明本例使用的各个属性：

```

Properties {
    _MainTex ("Base (RGB)", 2D) = "white" {}
    _Bloom ("Bloom (RGB)", 2D) = "black" {}
}

```

```
_LuminanceThreshold ("Luminance Threshold", Float) = 0.5
_BloomSize ("Blur Size", Float) = 1.0
}
```

`_MainTex`对应了输入的渲染纹理。`_Bloom`是高斯模糊后的较亮区域，`_LuminanceThreshold`是用于提取较亮区域使用的阈值，而`_BlurSize`和12.4节中的作用相同，用于控制不同迭代之间高斯模糊的模糊区域范围。

(2) 在本节中，我们仍然使用`CGINCLUDE`来组织代码。我们在`SubShader`块中利用`CGINCLUDE`和`ENDCG`语义来定义一系列代码：

```
SubShader {
    CGINCLUDE
    ...
    ENDCG
    ...
}
```

(3) 声明代码中需要使用的各个变量：

```
sampler2D _MainTex;
half4 _MainTex_TexelSize;
sampler2D _Bloom;
float _LuminanceThreshold;
float _BlurSize;
```

(4) 我们首先定义提取较亮区域需要使用的顶点着色器和片元着色器：

```
struct v2f {
    float4 pos : SV_POSITION;
    half2 uv : TEXCOORD0;
};

v2f vertExtractBright(appdata_img v) {
    v2f o;

    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);
}
```

```

        o.uv = v.texcoord;

        return o;
    }

    fixed luminance(fixed4 color) {
        return 0.2125 * color.r + 0.7154 * color.g + 0.0721 * color.b;
    }

    fixed4 fragExtractBright(v2f i) : SV_Target {
        fixed4 c = tex2D(_MainTex, i.uv);
        fixed val = clamp(luminance(c) - _LuminanceThreshold, 0.0,
1.0);

        return c * val;
    }

```

顶点着色器和之前的实现完全相同。在片元着色器中，我们将采样得到的亮度值减去阈值\_LuminanceThreshold，并把结果截取到0~1范围内。然后，我们把该值和原像素值相乘，得到提取后的亮部区域。

(5) 然后，我们定义了混合亮部图像和原图像时使用的顶点着色器和片元着色器：

```

struct v2fBloom {
    float4 pos : SV_POSITION;
    half4 uv : TEXCOORD0;
};

v2fBloom vertBloom(appdata_img v) {
    v2fBloom o;

    o.pos = mul (UNITY_MATRIX_MVP, v.vertex);
    o.uv.xy = v.texcoord;
    o.uv.zw = v.texcoord;

    #if UNITY_UV_STARTS_AT_TOP
    if (_MainTex_TexelSize.y < 0.0)
        o.uv.w = 1.0 - o.uv.w;
    #endif

    return o;
}

```



```
fixed4 fragBloom(v2fBloom i) : SV_Target {
    return tex2D(_MainTex, i.uv.xy) + tex2D(_Bloom, i.uv.zw);
}
```

这里使用的顶点着色器与之前的有所不同，我们定义了两个纹理坐标，并存储在同一个类型为`half4`的变量`uv`中。它的`xy`分量对应了`_MainTex`，即原图像的纹理坐标。而它的`zw`分量是`_Bloom`，即模糊后的较亮区域的纹理坐标。我们需要对这个纹理坐标进行平台差异化处理（详见5.6.1节）。

片元着色器的代码就很简单了。我们只需要把两张纹理的采样结果相加混合即可。

（6）接着，我们定义了Bloom效果需要的4个Pass:

```
ZTest Always Cull Off ZWrite Off

Pass {
    CGPROGRAM
    #pragma vertex vertExtractBright
    #pragma fragment fragExtractBright

    ENDCG
}

UsePass "Unity Shaders Book/Chapter 12/Gaussian
Blur/GAUSSIAN_BLUR_VERTICAL"

UsePass "Unity Shaders Book/Chapter 12/Gaussian
Blur/GAUSSIAN_BLUR_HORIZONTAL"

Pass {
    CGPROGRAM
    #pragma vertex vertBloom
    #pragma fragment fragBloom

    ENDCG
}
```

其中，第二个和第三个Pass我们直接使用了12.4节高斯模糊中定义的两个Pass，这是通过UsePass语义指明它们的Pass名来实现的。需要注意的是，由于Unity内部会把所有Pass的Name转换成大写字母表示，因此在使用UsePass命令时我们必须使用大写形式的名字。

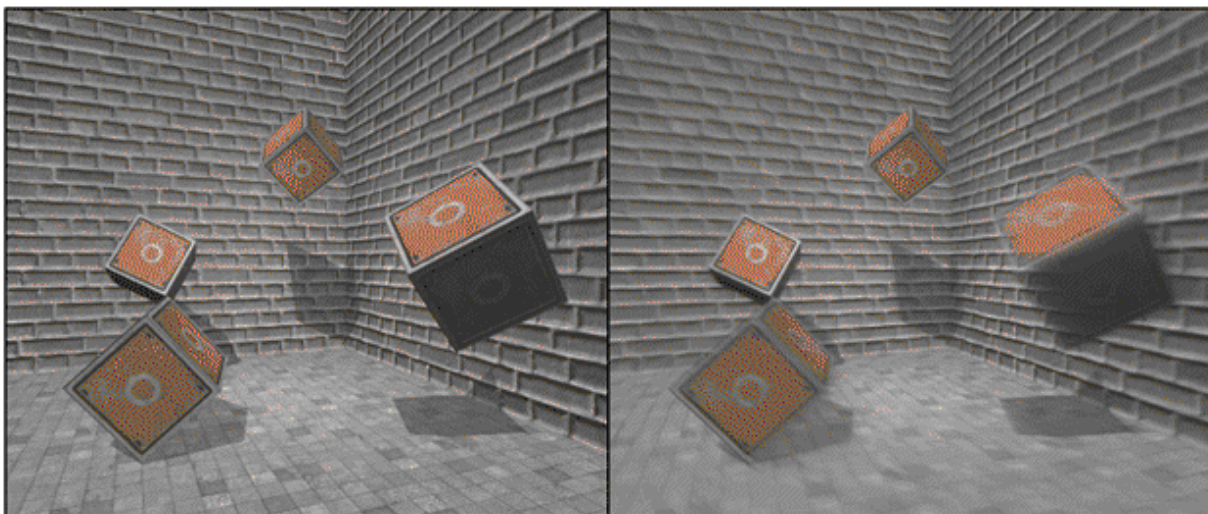
(7) 最后，我们关闭了该Shader的Fallback:

Fallback Off
--------------

完成后返回编辑器，并把Chapter12-Bloom拖曳到摄像机的Bloom.cs脚本中的bloomShader参数中。当然，我们可以在Bloom.cs的脚本面板中将bloomShader参数的默认值设置为Chapter12-Bloom，这样就不需要以后使用时每次都手动拖曳了。

## 12.6 运动模糊

运动模糊是真实世界中的摄像机的一种效果。如果在摄像机曝光时，拍摄场景发生了变化，就会产生模糊的画面。运动模糊在我们的日常生活中是非常常见的，只要留心观察，就可以发现无论是体育报道还是各个电影里，都有运动模糊的身影。运动模糊效果可以让物体运动看起来更加真实平滑，但在计算机产生的图像中，由于不存在曝光这一物理现象，渲染出来的图像往往都棱角分明，缺少运动模糊。在一些诸如赛车类型的游戏中，为画面添加运动模糊是一种常见的处理方法。在这一节中，我们将学习如何在屏幕后处理中实现运动模糊的效果。在本节结束后，我们将得到类似图12.11中的效果。



▲ 图12.11 左边为原效果，右边为应用运动模糊后的效果

运动模糊的实现有多种方法。一种实现方法是利用一块**累积缓存（accumulation buffer）**来混合多张连续的图像。当物体快速移动产生多张图像后，我们取它们之间的平均值作为最后的运动模糊图像。然而，这种暴力的方法对性能消耗很大，因为想要获取多张帧图像往往意味着我们需要在同一帧里渲染多次场景。另一种应用广泛的方法是创建和使用**速度缓存（velocity buffer）**，这个缓存中存储了各个像素当前的运动速度，然后利用该值来决定模糊的方向和大小。

在本节中，我们将使用类似上述第一种方法的实现来模拟运动模糊的效果。我们不需要在一帧中把场景渲染多次，但需要保存之前的渲染结果，不断把当前的渲染图像叠加到之前的渲染图像中，从而产生一种运动轨迹的视觉效果。这种方法与原始的利用累计缓存的方法相比性能更好，但模糊效果可能会略有影响。

为此，我们需要进行如下准备工作。

(1) 新建一个场景。在本书资源中，该场景名为Scene\_12\_6。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window → Lighting → Skybox中去掉场景中的天空盒子。

(2) 我们需要搭建一个测试运动模糊的场景。在本书资源的实现中，我们构建了一个包含3面墙的房间，并放置了4个立方体，它们均使用了我们在9.5节中创建的标准材质。同时，我们把本书资源中的Translating.cs脚本拖曳给摄像机，让其在场景中不断运动。

(3) 新建一个脚本。在本书资源中，该脚本名为MotionBlur.cs。把该脚本拖曳到摄像机上。

(4) 新建一个Unity Shader。在本书资源中，该Shader名为Chapter12-MotionBlur。

我们首先来编写MotionBlur.cs脚本。打开该脚本，并进行如下修改。

(1) 首先，继承12.1节中创建的基类：

```
public class MotionBlur : PostEffectsBase {
```

(2) 声明该效果需要的Shader，并据此创建相应的材质：

```
public Shader motionBlurShader;
private Material motionBlurMaterial = null;

public Material material {
    get {
        motionBlurMaterial =
        CheckShaderAndCreateMaterial(motionBlurShader, motionBlurMaterial);
```

```
        return motionBlurMaterial;
    }
}
```

(3) 定义运动模糊在混合图像时使用的模糊参数:

```
[Range(0.0f, 0.9f)]
public float blurAmount = 0.5f;
```

**blurAmount**的值越大，运动拖尾的效果就越明显，为了防止拖尾效果完全替代当前帧的渲染结果，我们把它的值截取在0.0~0.9范围内。

(4) 定义一个**RenderTexture**类型的变量，保存之前图像叠加的结果:

```
private RenderTexture accumulationTexture;

void OnDisable() {
    DestroyImmediate(accumulationTexture);
}
```

在上面的代码里，我们在该脚本不运行时，即调用**OnDisable**函数时，立即销毁**accumulation Texture**。这是因为，我们希望在下一次开始应用运动模糊时重新叠加图像。

(5) 最后，我们需要定义运动模糊使用的**OnRenderImage**函数:

```
void OnRenderImage (RenderTexture src, RenderTexture dest) {
    if (material != null) {
        // Create the accumulation texture
        if (accumulationTexture == null ||
            accumulationTexture.width != src.width ||
            accumulationTexture.height != src.height) {
            DestroyImmediate(accumulationTexture);
            accumulationTexture = new RenderTexture(src.width,
src.height, 0);
            accumulationTexture.hideFlags =
```

```
HideFlags.HideAndDontSave;
        Graphics.Blit(src, accumulationTexture);
    }

    // We are accumulating motion over frames without
clear/discard
    // by design, so silence any performance warnings from
Unity
    accumulationTexture.MarkRestoreExpected();

    material.SetFloat("_BlurAmount", 1.0f - blurAmount);

    Graphics.Blit (src, accumulationTexture, material);
    Graphics.Blit (accumulationTexture, dest);
} else {
    Graphics.Blit(src, dest);
}
}
```

在确认材质可用后，我们首先判断用于混合图像的 `accumulationTexture` 是否满足条件。我们不仅判断它是否为空，还判断它是否与当前的屏幕分辨率相等，如果不满足，就说明我们需要重新创建一个适合于当前分辨率的 `accumulationTexture` 变量。创建完毕后，由于我们会自己控制该变量的销毁，因此可以把它的 `hideFlags` 设置为 `HideFlags.HideAndDontSave`，这意味着这个变量不会显示在 `Hierarchy` 中，也不会保存到场景中。然后，我们使用当前的帧图像初始化 `accumulation Texture`（使用 `Graphics.Blit(src, accumulationTexture)` 代码）。

当得到了有效的 `accumulationTexture` 变量后，我们调用了 `accumulationTexture.Mark RestoreExpected` 函数来表明我们需要进行一个渲染纹理的恢复操作。**恢复操作（restore operation）** 发生在渲染到纹理而该纹理又没有被提前清空或销毁的情况下。在本例中，我们每次调用 `OnRenderImage` 时都需要把当前的帧图像和 `accumulationTexture` 中的图像混合，`accumulationTexture` 纹理不需要提前清空，因为它保存了

我们之前的混合结果。然后，我们将参数传递给材质，并调用 `Graphics.Blit (src, accumulationTexture, material)` 把当前的屏幕图像 `src` 叠加到 `accumulationTexture` 中。最后使用 `Graphics.Blit (accumulationTexture, dest)` 把结果显示到屏幕上。

下面，我们来实现 `Shader` 的部分。本节实现的运动模糊非常简单，我们打开 `Chapter12-MotionBlur`，进行如下修改。

(1) 我们首先需要声明本例使用的各个属性：

```
Properties {  
    _MainTex ("Base (RGB)", 2D) = "white" {}  
    _BlurAmount ("Blur Amount", Float) = 1.0  
}
```

`_MainTex` 对应了输入的渲染纹理。`_BlurAmount` 是混合图像时使用的混合系数。

(2) 在本节中，我们使用 `CGINCLUDE` 来组织代码。我们在 `SubShader` 块中利用 `CGINCLUDE` 和 `ENDCG` 语义来定义一系列代码：

```
SubShader {  
    CGINCLUDE  
    ...  
    ENDCG  
    ...  
}
```

(3) 声明代码中需要使用的各个变量：

```
sampler2D _MainTex;  
fixed _BlurAmount;
```

(4) 顶点着色器的代码与之前章节使用的代码完全一样：



```
struct v2f {
    float4 pos : SV_POSITION;
    half2 uv : TEXCOORD0;
};

v2f vert(appdata_img v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.uv = v.texcoord;

    return o;
}
```

(5) 下面，我们定义了两个片元着色器，一个用于更新渲染纹理的RGB通道部分，另一个用于更新渲染纹理的A通道部分：

```
fixed4 fragRGB (v2f i) : SV_Target {
    return fixed4(tex2D(_MainTex, i.uv).rgb, _BlurAmount);
}

half4 fragA (v2f i) : SV_Target {
    return tex2D(_MainTex, i.uv);
}
```

RGB通道版本的Shader对当前图像进行采样，并将其A通道的值设为`_BlurAmount`，以便在后面混合时可以使用它的透明通道进行混合。A通道版本的代码就更简单了，直接返回采样结果。实际上，这个版本只是为了维护渲染纹理的透明通道值，不使其受到混合时使用的透明度值的影响。

(6) 然后，我们定义了运动模糊所需的Pass。在本例中我们需要两个Pass，一个用于更新渲染纹理的RGB通道，另一个用于更新A通道。之所以要把A通道和RGB通道分开，是因为在更新RGB时我们需要设置它的A通道来混合图像，但又不希望A通道的值写入渲染纹理中。

```

ZTest Always Cull Off ZWrite Off

Pass {
    Blend SrcAlpha OneMinusSrcAlpha
    ColorMask RGB

    CGPROGRAM

    #pragma vertex vert
    #pragma fragment fragRGB

    ENDCG
}

Pass {
    Blend One Zero
    ColorMask A

    CGPROGRAM

    #pragma vertex vert
    #pragma fragment fragA

    ENDCG
}

```

(7) 最后，我们关闭了Shader的Fallback:

```
Fallback Off
```

完成后返回编辑器，并把Chapter12-MotionBlur拖曳到摄像机的MotionBlur.cs脚本中的motionBlurShader参数中。当然，我们可以在MotionBlur.cs的脚本面板中将motionBlurShader参数的默认值设置为Chapter12-MotionBlur，这样就不需要以后使用时每次都手动拖曳了。

本节是对运动模糊的一种简单实现。我们混合了连续帧之间的图像，这样得到一张具有模糊拖尾的图像。然而，当物体运动速度过快时，这种方法可能会造成单独的帧图像变得可见。在第13章中，我们会学习如何利用深度纹理重建速度来模拟运动模糊效果。

## 12.7 扩展阅读

本章介绍了如何在Unity中利用渲染纹理实现屏幕后处理效果，并且介绍了几种常见的屏幕特效的实现方法。这些效果都使用了图像处理中的一些算法，以达到特定的图像效果。除了本章介绍的这些效果外，读者可以在Unity的Image Effect

(<http://docs.unity3d.com/Manual/comp-ImageEffects.html>) 包中找到更多特效的实现。在GPU Gems系列 (<https://developer.nvidia.com/gpugems/GPUGems>) 中，也介绍了许多基于图像处理的渲染技术。例如，《GPU Gems 3》的第27章，介绍了一种景深效果的实现方法。除此之外，读者也可以在Unity的资源商店和其他网络资源中找到许多出色的屏幕特效。

## 第13章 使用深度和法线纹理

在第12章中，我们学习的屏幕后处理效果都只是在屏幕颜色图像上进行各种操作来实现的。然而，很多时候我们不仅需要当前屏幕的颜色信息，还希望得到深度和法线信息。例如，在进行边缘检测时，直接利用颜色信息会使检测到的边缘信息受物体纹理和光照等外部因素的影响，得到很多我们不需要的边缘点。一种更好的方法是，我们可以在深度纹理和法线纹理上进行边缘检测，这些图像不会受纹理和光照的影响，而仅仅保存了当前渲染物体的模型信息，通过这样的方式检测出来的边缘更加可靠。

在本章中，我们将学习如何在Unity中获取深度纹理和法线纹理来实现特定的屏幕后处理效果。在13.1节中，我们首先会学习如何在Unity中获取这两种纹理。在13.2节中，我们会利用深度纹理来计算摄像机的移动速度，实现摄像机的运动模糊效果。在13.3节中，我们会学习如何利用深度纹理来重建屏幕像素在世界空间中的位置，从而模拟屏幕雾效。13.4节会再次学习边缘检测的另一种实现，即利用深度和法线纹理进行边缘检测。

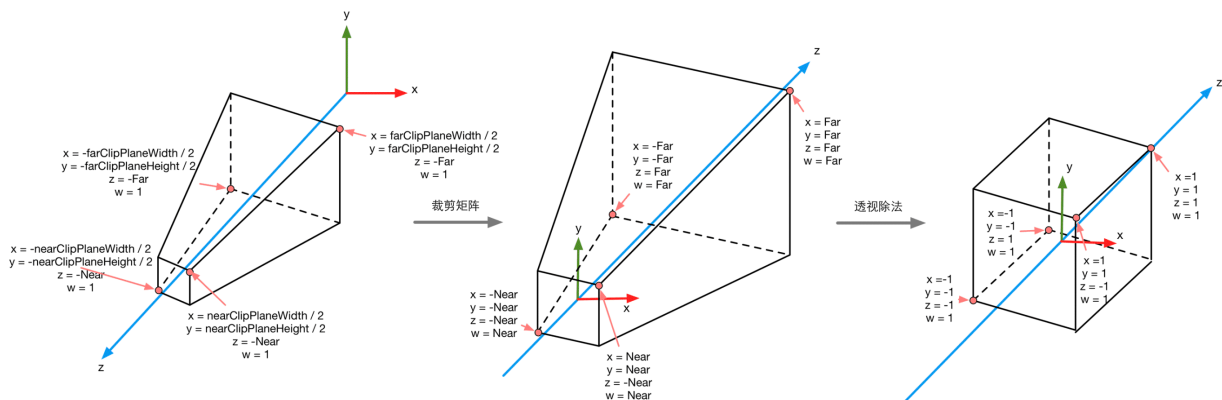
### 13.1 获取深度和法线纹理

虽然在Unity里获取深度和法线纹理的代码非常简单，但是我们有必要在这之前首先了解它们背后的实现原理。

### 13.1.1 背后的原理

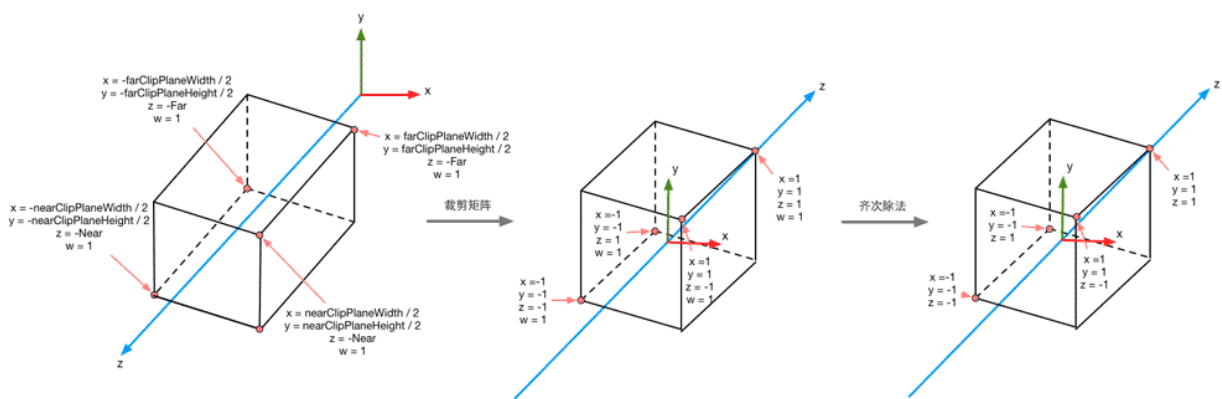
深度纹理实际就是一张渲染纹理，只不过它里面存储的像素值不是颜色值，而是一个高精度的深度值。由于被存储在一张纹理中，深度纹理里的深度值范围是 $[0, 1]$ ，而且通常是非线性分布的。那么，这些深度值是从哪里得到的呢？要回答这个问题，我们需要回顾在第4章学过的顶点变换的过程。总体来说，这些深度值来自于顶点变换后得到的归一化的设备坐标（Normalized Device Coordinates，NDC）。回顾一下，一个模型要想最终被绘制在屏幕上，需要把它的顶点从模型空间变换到齐次裁剪坐标系下，这是通过在顶点着色器中乘以MVP变换矩阵得到的。在变换的最后一步，我们需要使用一个投影矩阵来变换顶点，当我们使用的是透视投影类型的摄像机时，这个投影矩阵就是非线性的，具体过程可回顾4.6.7小节。

图13.1显示了4.6.7小节中给出的Unity中透视投影对顶点的变换过程。图13.1中最左侧的图显示了投影变换前，即观察空间下视锥体的结构及相应的顶点位置，中间的图显示了应用透视裁剪矩阵后的变换结果，即顶点着色器阶段输出的顶点变换结果，最右侧的图则是底层硬件进行了透视除法后得到的归一化的设备坐标。需要注意的是，这里的投影过程是建立在Unity对坐标系的假定上的，也就是说，我们针对的是观察空间为右手坐标系，使用列矩阵在矩阵右侧进行相乘，且变换到NDC后 $z$ 分量范围将在 $[-1, 1]$ 之间的情况。而在类似DirectX这样的图形接口中，变换后 $z$ 分量范围将在 $[0, 1]$ 之间。如果需要在其他图形接口下实现本章的类似效果，需要对一些计算参数做出相应变化。关于变换时使用的矩阵运算，读者可以参考4.6.7小节。



▲ 图13.1 在透视投影中，投影矩阵首先对顶点进行了缩放。在经过齐次除法后，透视投影的裁剪空间会变换到一个立方体。图中标注了4个关键点经过投影矩阵变换后的结果

图13.2显示了在使用正交摄像机时投影变换的过程。同样，变换后会得到一个范围为 $[-1, 1]$ 的立方体。正交投影使用的变换矩阵是线性的。



▲ 图13.2 在正交投影中，投影矩阵对顶点进行了缩放。在经过齐次除法后，正交投影的裁剪空间会变换到一个立方体。图中标注了4个关键点经过投影矩阵变换后的结果

在得到NDC后，深度纹理中的像素值就可以很方便地计算得到了，这些深度值就对应了NDC中顶点坐标的 $z$ 分量的值。由于NDC中 $z$

分量的范围在 $[-1, 1]$ ，为了让这些值能够存储在一张图像中，我们需要使用下面的公式对其进行映射：

$$d=0.5 \cdot z_{ndc}+0.5$$

其中， $d$ 对应了深度纹理中的像素值， $z_{ndc}$ 对应了NDC坐标中的 $z$ 分量的值。

那么Unity是怎么得到这样一张深度纹理的呢？在Unity中，深度纹理可以直接来自于真正的深度缓存，也可以是由一个单独的Pass渲染而得，这取决于使用的渲染路径和硬件。通常来讲，当使用延迟渲染路径（包括遗留的延迟渲染路径）时，深度纹理理所当然可以访问到，因为延迟渲染会把这些信息渲染到G-buffer中。而当无法直接获取深度缓存时，深度和法线纹理是通过一个单独的Pass渲染而得的。具体实现是，Unity会使用着色器替换（Shader Replacement）技术选择那些渲染类型（即SubShader的RenderType标签）为Opaque的物体，判断它们使用的渲染队列是否小于等于2 500（内置的Background、Geometry和AlphaTest渲染队列均在此范围内），如果满足条件，就把它渲染到深度和法线纹理中。因此，要想让物体能够出现在深度和法线纹理中，就必须在Shader中**设置正确的RenderType标签**。

在Unity中，我们可以选择让一个摄像机生成一张深度纹理或是一张深度+法线纹理。当选择前者，即只需要一张单独的深度纹理时，Unity会直接获取深度缓存或是按之前讲到的着色器替换技术，选取需要的不透明物体，并使用它投射阴影时使用的Pass（即LightMode被设置为ShadowCaster的Pass，详见9.4节）来得到深度纹理。如果Shader中不包含这样一个Pass，那么这个物体就不会出现在深度纹理中（当然，



它也不能向其他物体投射阴影)。深度纹理的精度通常是24位或16位,这取决于使用的深度缓存的精度。如果选择生成一张深度+法线纹理,Unity会创建一张和屏幕分辨率相同、精度为32位(每个通道为8位)的纹理,其中观察空间下的法线信息会被编码进纹理的R和G通道,而深度信息会被编码进B和A通道。法线信息的获取在延迟渲染中是可以非常容易就得到的,Unity只需要合并深度和法线缓存即可。而在前向渲染中,默认情况下是不会创建法线缓存的,因此Unity底层使用了一个单独的Pass把整个场景再次渲染一遍来完成。这个Pass被包含在Unity内置的一个Unity Shader中,我们可以在内置的builtin\_shaders-xxx/DefaultResources/Camera-DepthNormalTexture.shader文件中找到这个用于渲染深度和法线信息的Pass。

### 13.1.2 如何获取

在Unity中,获取深度纹理是非常简单的,我们只需要告诉Unity:“嘿,把深度纹理给我!”然后再在Shader中直接访问特定的纹理属性即可。这个与Unity沟通的过程是通过在脚本中设置摄像机的depthTextureMode来完成的,例如我们可以通过下面的代码来获取深度纹理:

```
camera.depthTextureMode = DepthTextureMode.Depth;
```

一旦设置好了上面的摄像机模式后,我们就可以在Shader中通过声明\_CameraDepthTexture变量来访问它。这个过程非常简单,但我们需要知道这两行代码的背后,Unity为我们做了许多工作(见13.1.1节)。

同理,如果想要获取深度+法线纹理,我们只需要在代码中这样设置:

```
camera.depthTextureMode = DepthTextureMode.DepthNormals;
```

然后在Shader中通过声明\_CameraDepthNormalsTexture变量来访问它。

我们还可以组合这些模式，让一个摄像机同时产生一张深度和深度+法线纹理：

```
camera.depthTextureMode |= DepthTextureMode.Depth;  
camera.depthTextureMode |= DepthTextureMode.DepthNormals;
```

在Unity 5中，我们还可以在摄像机的Camera组件上看到当前摄像机是否需要渲染深度或深度+法线纹理。当在Shader中访问到深度纹理\_CameraDepthTexture后，我们就可以使用当前像素的纹理坐标对它进行采样。绝大多数情况下，我们直接使用tex2D函数采样即可，但在某些平台（例如PS3和PSP2）上，我们需要一些特殊处理。Unity为我们提供了一个统一的宏SAMPLE\_DEPTH\_TEXTURE，用来处理这些由于平台差异造成的问题。而我们只需要在Shader中使用SAMPLE\_DEPTH\_TEXTURE宏对深度纹理进行采样，例如：

```
float d = SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, i.uv);
```

其中，i.uv是一个float2类型的变量，对应了当前像素的纹理坐标。类似的宏还有SAMPLEDEPTH\_TEXTURE\_PROJ和SAMPLE\_DEPTH\_TEXTURE\_LOD。SAMPLE\_DEPTH\_TEXTURE\_PROJ宏同样接受两个参数—深度纹理和一个float3或float4类型的纹理坐标，它的内部使用了tex2Dproj这样的函数进行投影纹理采样，纹理坐标的前两个分量首先会除以最后一个分量，再进行纹理采样。如果提供了第四个分量，还会进行一次比较，通常用于阴影的实现中。

SAMPLE\_DEPTH\_TEXTURE\_PROJ的第二个参数通常是由顶点着色器输出插值而得的屏幕坐标，例如：

```
float d = SAMPLE_DEPTH_TEXTURE_PROJ(_CameraDepthTexture,  
UNITY_PROJ_COORD(i.scrPos));
```

其中，i.scrPos是在顶点着色器中通过调用ComputeScreenPos(o.pos)得到的屏幕坐标。上述这些宏的定义，读者可以在Unity内置的HLSLSupport.cginc文件中找到。

当通过纹理采样得到深度值后，这些深度值往往是非线性的，这种非线性来自于透视投影使用的裁剪矩阵。然而，在我们的计算过程中通常是需要线性的深度值，也就是说，我们需要把投影后的深度值变换到线性空间下，例如视角空间下的深度值。那么，我们应该如何进行这个转换呢？实际上，我们只需要倒推顶点变换的过程即可。下面我们以透视投影为例，推导如何由深度纹理中的深度信息计算得到视角空间下的深度值。

由4.6.7节可知，当我们使用透视投影的裁剪矩阵 $\mathbf{P}_{clip}$ 对视角空间下的一个顶点进行变换后，裁剪空间下顶点的z和w分量为：

$$z_{clip} = -z_{view} \frac{Far + Near}{Far - Near} - \frac{2 \cdot Near \cdot Far}{Far - Near}$$
$$w_{clip} = -z_{view}$$

其中， $Far$ 和 $Near$ 分别是远近裁剪平面的距离。然后，我们通过齐次除法就可以得到NDC下的z分量：

$$z_{ndc} = \frac{z_{clip}}{w_{clip}} = \frac{Far + Near}{Far - Near} + \frac{2 \cdot Near \cdot Far}{(Far - Near) \cdot z_{view}}$$

在13.1.1节中我们知道，深度纹理中的深度值是通过下面的公式由NDC计算而得的：

$$d = 0.5 \cdot z_{ndc} + 0.5$$

由上面的这些式子，我们可以推导出用 $d$ 表示而得的 $z_{view}$ 的表达式：

$$z_{view} = \frac{1}{\frac{Far - Near}{Near \cdot Far} d - \frac{1}{Near}}$$

由于在Unity使用的视角空间中，摄像机正向对应的 $z$ 值均为负值，因此为了得到深度值的正数表示，我们需要对上面的结果取反，最后得到的结果如下：

$$z'_{view} = \frac{1}{\frac{Near - Far}{Near \cdot Far} d + \frac{1}{Near}}$$

它的取值范围就是视锥体深度范围，即 $[Near, Far]$ 。如果我们想得到范围在 $[0, 1]$ 之间的深度值，只需要把上面得到的结果除以 $Far$ 即可。

这样，0就表示该点与摄像机位于同一位置，1表示该点位于视锥体的远裁剪平面上。结果如下：

$$z_{01} = \frac{1}{\frac{Near - Far}{Near}d + \frac{Far}{Near}}$$

幸运的是，Unity提供了两个辅助函数来为我们进行上述的计算过程—LinearEyeDepth和Linear01Depth。LinearEyeDepth负责把深度纹理的采样结果转换到视角空间下的深度值，也就是我们上面得到的 $z'_{view}$ 。而Linear01Depth则会返回一个范围在[0, 1]的线性深度值，也就是我们上面得到的 $z_{01}$ 。这两个函数内部使用了内置的\_ZBufferParams变量来得到远近裁剪平面的距离。

如果我们需要获取深度+法线纹理，可以直接使用tex2D函数对\_CameraDepthNormalsTexture进行采样，得到里面存储的深度和法线信息。Unity提供了辅助函数来为我们对这个采样结果进行解码，从而得到深度值和法线方向。这个函数是DecodeDepthNormal，它在UnityCG.cginc里被定义：

```
inline void DecodeDepthNormal( float4 enc, out float depth, out
float3 normal )
{
    depth = DecodeFloatRG (enc.zw);
    normal = DecodeViewNormalStereo (enc);
}
```

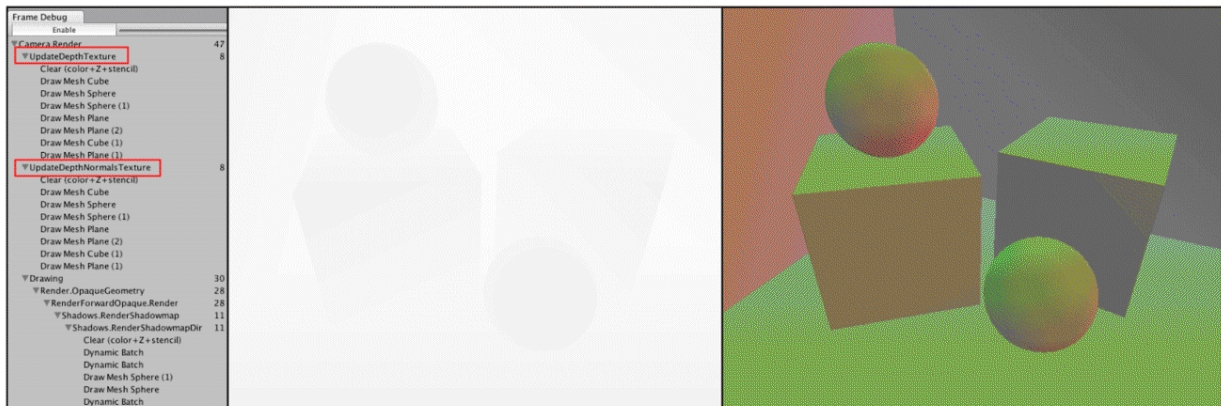
DecodeDepthNormal的第一个参数是对深度+法线纹理的采样结果，这个采样结果是Unity对深度和法线信息编码后的结果，它的xy分量存储的是视角空间下的法线信息，而深度信息被编码进了zw分量。通过调用DecodeDepthNormal函数对采样结果解码后，我们就可以得到

解码后的深度值和法线。这个深度值是范围在[0, 1]的线性深度值（这与单独的深度纹理中存储的深度值不同），而得到的法线则是视角空间下的法线方向。同样，我们也可以通过调用DecodeFloatRG和DecodeViewNormalStereo来解码深度+法线纹理中的深度和法线信息。

至此，我们已经学会了如何在Unity里获取及使用深度和法线纹理。下面，我们会学习如何使用它们实现各种屏幕特效。

### 13.1.3 查看深度和法线纹理

很多时候，我们希望可以查看生成的深度和法线纹理，以便对Shader进行调试。Unity 5提供了一个方便的方法来查看摄像机生成的深度和法线纹理，这个方法就是利用帧调试器（Frame Debugger）。图13.3显示了使用帧调试器查看到的深度纹理和深度+法线纹理。



▲ 图13.3 使用Frame Debugger查看深度纹理（左）和深度+法线纹理（右）。如果当前摄像机需要生成深度和法线纹理，帧调试器的面板中就会出现相应的渲染事件。只要单击对应的事件就可以查看得到的深度和法线纹理

使用帧调试器查看到的深度纹理是非线性空间的深度值，而深度+法线纹理都是由Unity编码后的结果。有时，显示出线性空间下的深度

信息或解码后的法线方向会更加有用。此时，我们可以自行在片元着色器中输出转换或解码后的深度和法线值，如图13.4所示。输出代码非常简单，我们可以使用类似下面的代码来输出线性深度值：

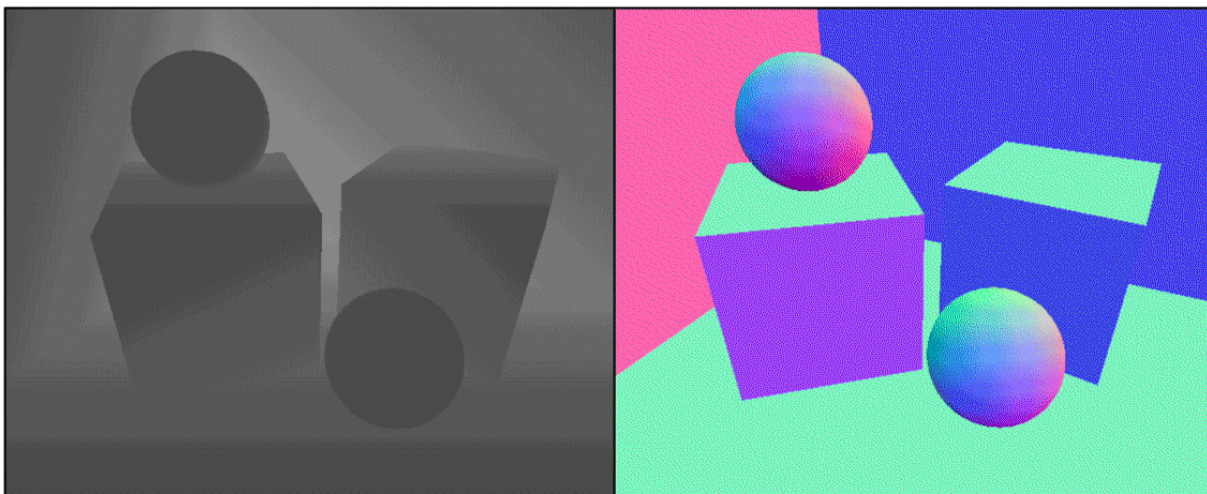
```
float depth = SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, i.uv);  
float linearDepth = Linear01Depth(depth);  
return fixed4(linearDepth, linearDepth, linearDepth, 1.0);
```

或是输出法线方向：

```
fixed3 normal =  
DecodeViewNormalStereo(tex2D(_CameraDepthNormalsTexture, i.uv).xy);  
return fixed4(normal * 0.5 + 0.5, 1.0);
```

在查看深度纹理时，读者得到的画面有可能几乎是全黑或全白的。这时候读者可以把摄像机的远裁剪平面的距离（Unity默认为1000）调小，使视锥体的范围刚好覆盖场景的所在区域。这是因为，由于投影变换时需要覆盖从近裁剪平面到远裁剪平面的所有深度区域，当远裁剪平面的距离过大时，会导致离摄像机较近的距离被映射到非常小的深度值，如果场景是一个封闭的区域（如图13.4所示），那么这就会导致画面看起来几乎是全黑的。相反，如果场景是一个开放区域，且物体离摄像机的距离较远，就会导致画面几乎是全白的。





▲ 图13.4 左边：线性空间下的深度纹理。右边：解码后并且被映射到[0, 1]范围内的视角空间下的法线纹理

## 13.2 再谈运动模糊

在12.6节中，我们学习了如何通过混合多张屏幕图像来模拟运动模糊的效果。但是，另一种应用更加广泛的技术则是使用速度映射图。速度映射图中存储了每个像素的速度，然后使用这个速度来决定模糊的方向和大小。速度缓冲的生成有多种方法，一种方法是把场景中所有物体的速度渲染到一张纹理中。但这种方法的缺点在于需要修改场景中所有物体的Shader代码，使其添加计算速度的代码并输出到一个渲染纹理中。

《GPU Gems3》在第27章

([http://http.developer.nvidia.com/GPUGems3/gpugems3\\_ch27.html](http://http.developer.nvidia.com/GPUGems3/gpugems3_ch27.html)) 介绍了一种生成速度映射图的方法。这种方法利用深度纹理在片元着色器中为每个像素计算其在世界空间下的位置，这是通过使用当前的视角投影矩阵的逆矩阵对NDC下的顶点坐标进行变换得到的。当得到世

界空间中的顶点坐标后，我们使用前一帧的视角投影矩阵对其进行变换，得到该位置在前一帧中的NDC坐标。然后，我们计算前一帧和当前帧的位置差，生成该像素的速度。这种方法的优点是可以在一个屏幕后处理步骤中完成整个效果的模拟，但缺点是需要在片元着色器中进行两次矩阵乘法的操作，对性能有所影响。

为了使用深度纹理模拟运动模糊，我们需要进行如下准备工作。

(1) 新建一个场景。在本书资源中，该场景名为Scene\_13\_2。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window → Lighting → Skybox中去掉场景中的天空盒子。

(2) 我们需要搭建一个测试运动模糊的场景。在本书资源的实现中，我们构建了一个包含3面墙的房间，并放置了4个立方体，它们都使用了我们在9.5节中创建的标准材质。同时，我们把本书资源中的Translating.cs脚本拖曳给摄像机，让其在场景中不断运动。

(3) 新建一个脚本。在本书资源中，该脚本名为MotionBlurWithDepthTexture.cs。把该脚本拖曳到摄像机上。

(4) 新建一个Unity Shader。在本书资源中，该Shader名为Chapter13-MotionBlurWithDepthTexture。

我们首先来编写MotionBlurWithDepthTexture.cs脚本。打开该脚本，并进行如下修改。

(1) 首先，继承12.1节中创建的基类：

```
public class MotionBlurWithDepthTexture : PostEffectsBase {
```

(2) 声明该效果需要的Shader，并据此创建相应的材质：

```
public Shader motionBlurShader;  
private Material motionBlurMaterial = null;  
public Material material {  
    get {  
        motionBlurMaterial =  
CheckShaderAndCreateMaterial(motionBlurShader, motionBlurMaterial);  
        return motionBlurMaterial;  
    }  
}
```

(3) 定义运动模糊时模糊图像使用的大小：

```
[Range(0.0f, 1.0f)]  
public float blurSize = 0.5f;
```

(4) 由于本节需要得到摄像机的视角和投影矩阵，我们需要定义一个Camera类型的变量，以获取该脚本所在的摄像机组件：

```
private Camera myCamera;  
public Camera camera {  
    get {  
        if (myCamera == null) {  
            myCamera = GetComponent<Camera>();  
        }  
        return myCamera;  
    }  
}
```

(5) 我们还需要定义一个变量来保存上一帧摄像机的视角\*投影矩阵：

```
private Matrix4x4 previousViewProjectionMatrix;
```

(6) 由于本例需要获取摄像机的深度纹理，我们在脚本的 `OnEnable` 函数中设置摄像机的状态：

```
void OnEnable() {  
    camera.depthTextureMode |= DepthTextureMode.Depth;  
}
```

(7) 最后，我们实现了 `OnRenderImage` 函数：

```
void OnRenderImage (RenderTexture src, RenderTexture dest) {  
    if (material != null) {  
        material.SetFloat("_BlurSize", blurSize);  
        material.SetMatrix("_PreviousViewProjectionMatrix",  
previousViewProjectionMatrix);  
        Matrix4x4 currentViewProjectionMatrix =  
camera.projectionMatrix * camera.worldToCameraMatrix;  
        Matrix4x4 currentViewProjectionInverseMatrix =  
currentViewProjectionMatrix.inverse;  
        material.SetMatrix("_CurrentViewProjectionInverseMatrix",  
currentViewProjectionInverseMatrix);  
        previousViewProjectionMatrix = currentViewProjectionMatrix;  
        Graphics.Blit (src, dest, material);  
    } else {  
        Graphics.Blit(src, dest);  
    }  
}
```

上面的 `OnRenderImage` 函数很简单，我们首先需要计算和传递运动模糊使用的各个属性。本例需要使用两个变换矩阵——前一帧的视角投影矩阵以及当前帧的视角投影矩阵的逆矩阵。因此，我们通过调用 `camera.worldToCameraMatrix` 和 `camera.projectionMatrix` 来分别得到当前摄像机的视角矩阵和投影矩阵。对它们相乘后取逆，得到当前帧的视角\*投影矩阵的逆矩阵，并传递给材质。然后，我们把取逆前的结果存储在 `previousViewProjectionMatrix` 变量中，以便在下一帧时传递给材质的 `_PreviousViewProjectionMatrix` 属性。

下面，我们来实现Shader的部分。打开Chapter13-MotionBlurWithDepthTexture，进行如下修改。

(1) 我们首先需要声明本例使用的各个属性：

```
Properties {  
    _MainTex ("Base (RGB)", 2D) = "white" {}  
    _BlurSize ("Blur Size", Float) = 1.0  
}
```

`_MainTex`对应了输入的渲染纹理，`_BlurSize`是模糊图像时使用的参数。我们注意到，虽然在脚本里设置了材质的`_PreviousViewProjectionMatrix`和`_CurrentViewProjectionInverseMatrix`属性，但并没有在`Properties`块中声明它们。这是因为Unity没有提供矩阵类型的属性，但我们仍然可以在CG代码块中定义这些矩阵，并从脚本中设置它们。

(2) 在本节中，我们使用`CGINCLUDE`来组织代码。我们在`SubShader`块中利用`CGINCLUDE`和`ENDCG`语义来定义一系列代码：

```
SubShader {  
    CGINCLUDE  
    ...  
    ENDCG  
    ...  
}
```

(3) 声明代码中需要使用的各个变量：

```
sampler2D _MainTex;  
half4 _MainTex_TexelSize;  
sampler2D _CameraDepthTexture;  
float4x4 _CurrentViewProjectionInverseMatrix;  
float4x4 _PreviousViewProjectionMatrix;  
half _BlurSize;
```

在上面的代码中，除了定义在**Properties**声明的**\_MainTex**和**\_BlurSize**属性，我们还声明了其他三个变量。**\_CameraDepthTexture**是Unity传递给我们的深度纹理，而**\_CurrentViewProjectionInverseMatrix**和**\_PreviousViewProjectionMatrix**是由脚本传递而来的矩阵。除此之外，我们还声明了**\_MainTex\_TexelSize**变量，它对应了主纹理的纹素大小，我们需要使用该变量来对深度纹理的采样坐标进行平台差异化处理（详见5.6.1节）。

（4）顶点着色器的代码和之前使用多次的代码基本一致，只是增加了专门用于对深度纹理采样的纹理坐标变量：

```
struct v2f {
    float4 pos : SV_POSITION;
    half2 uv : TEXCOORD0;
    half2 uv_depth : TEXCOORD1;
};
v2f vert(appdata_img v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);
    o.uv = v.texcoord;
    o.uv_depth = v.texcoord;
    #if UNITY_UV_STARTS_AT_TOP
    if (_MainTex_TexelSize.y < 0)
        o.uv_depth.y = 1 - o.uv_depth.y;
    #endif
    return o;
}
```

由于在本例中，我们需要同时处理多张渲染纹理，因此在**DirectX**这样的平台上，我们需要处理平台差异导致的图像翻转问题。在上面的代码中，我们对深度纹理的采样坐标进行了平台差异化处理，以便在类似**DirectX**的平台上，在开启了抗锯齿的情况下仍然可以得到正确的结果。

（5）片元着色器是算法的重点所在：

```

fixed4 frag(v2f i) : SV_Target {
    // Get the depth buffer value at this pixel.
    float d = SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture,
i.uv_depth);
    // H is the viewport position at this pixel in the range -1 to
1.
    float4 H = float4(i.uv.x * 2 - 1, i.uv.y * 2 - 1, d * 2 - 1,
1);
    // Transform by the view-projection inverse.
    float4 D = mul(_CurrentViewProjectionInverseMatrix, H);
    // Divide by w to get the world position.
    float4 worldPos = D / D.w;
    // Current viewport position
    float4 currentPos = H;
    // Use the world position, and transform by the previous view-
projection matrix.
    float4 previousPos = mul(_PreviousViewProjectionMatrix,
worldPos);
    // Convert to nonhomogeneous points [-1,1] by dividing by w.
    previousPos /= previousPos.w;
    // Use this frame's position and last frame's to compute the
pixel velocity.
    float2 velocity = (currentPos.xy - previousPos.xy)/2.0f;
    float2 uv = i.uv;
    float4 c = tex2D(_MainTex, uv);
    uv += velocity * _BlurSize;
    for (int it = 1; it < 3; it++, uv += velocity * _BlurSize) {
        float4 currentColor = tex2D(_MainTex, uv);
        c += currentColor;
    }
    c /= 3;
    return fixed4(c.rgb, 1.0);
}

```

我们首先需要利用深度纹理和当前帧的视角投影矩阵的逆矩阵来求得该像素在世界空间下的坐标。过程开始于对深度纹理的采样，我们使用内置的SAMPLE\_DEPTH\_TEXTURE宏和纹理坐标对深度纹理进行采样，得到了深度值 $d$ 。由13.1.2节可知， $d$ 是由NDC下的坐标映射而来的。我们想要构建像素的NDC坐标 $H$ ，就需要把这个深度值重新映射回NDC。这个映射很简单，只需要使用原映射的反函数即可，即 $d * 2 - 1$ 。同样，NDC的 $xy$ 分量可以由像素的纹理坐标映射而来（NDC下的 $xyz$ 分量范围均为 $[-1, 1]$ ）。当得到NDC下的坐标 $H$ 后，我们就可以使用



当前帧的视角\*投影矩阵的逆矩阵对其进行变换，并把结果值除以它的w分量来得到世界空间下的坐标表示worldPos。

一旦得到了世界空间下的坐标，我们就可以使用前一帧的视角\*投影矩阵对它进行变换，得到前一帧在NDC下的坐标previousPos。然后，我们计算前一帧和当前帧在屏幕空间下的位置差，得到该像素的速度velocity。

当得到该像素的速度后，我们就可以使用该速度值对它的邻域像素进行采样，相加后取平均值得到一个模糊的效果。采样时我们还使用了\_BlurSize来控制采样距离。

(6) 然后，我们定义了运动模糊所需的Pass:

```
Pass {  
    ZTest Always Cull Off ZWrite Off  
    CGPROGRAM  
    #pragma vertex vert  
    #pragma fragment frag  
    ENDCG  
}
```

(7) 最后，我们关闭了shader的Fallback:

```
Fallback Off
```

完成后返回编辑器，并把Chapter13-MotionBlurWithDepthTexture拖曳到摄像机的MotionBlur WithDepthTexture.cs脚本中的motionBlurShader参数中。当然，我们可以在MotionBlurWithDepthTexture.cs的脚本面板中将motionBlurShader参数的默认值设置为

Chapter13-MotionBlur WithDepthTexture，这样就不需要以后使用时每次都手动拖曳了。

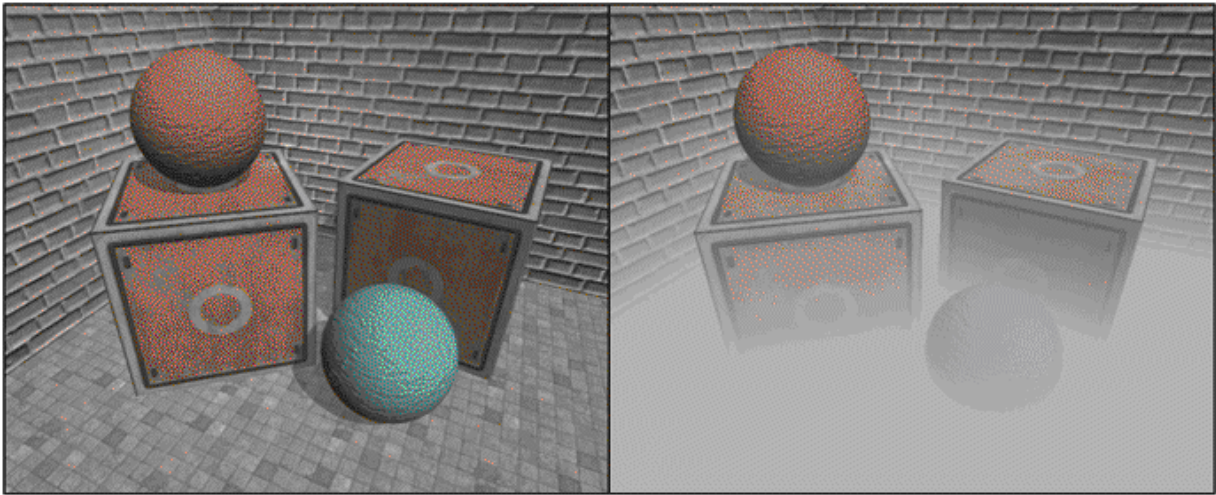
本节实现的运动模糊适用于场景静止、摄像机快速运动的情况，这是因为我们在计算时只考虑了摄像机的运动。因此，如果读者把本节中的代码应用到一个物体快速运动而摄像机静止的场景，会发现不会产生任何运动模糊效果。如果我们想要对快速移动的物体产生运动模糊的效果，就需要生成更加精确的速度映射图。读者可以在Unity自带的ImageEffect包中找到更多的运动模糊的实现方法。

本节选择在片元着色器中使用逆矩阵来重建每个像素在世界空间下的位置。但是，这种做法往往会影响性能，在13.3节中，我们会介绍一种更快速的由深度纹理重建世界坐标的方法。

## 13.3 全局雾效

**雾效（Fog）**是游戏里经常使用的一种效果。Unity内置的雾效可以产生基于距离的线性或指数雾效。然而，要想在自己编写的顶点/片元着色器中实现这些雾效，我们需要在Shader中添加`#pragma multi_compile_fog`指令，同时还需要使用相关的内置宏，例如`UNITY_FOG_COORDS`、`UNITY_TRANSFER_FOG`和`UNITY_APPLY_FOG`等。这种方法的缺点在于，我们不仅需要为场景中所有物体添加相关的渲染代码，而且能够实现的效果也非常有限。当我们需要对雾效进行一些个性化操作时，例如使用基于高度的雾效等，仅仅使用Unity内置的雾效就变得不再可行。

在本节中，我们将会学习一种基于屏幕后处理的全局雾效的实现。使用这种方法，我们不需要更改场景内渲染的物体所使用的Shader代码，而仅仅依靠一次屏幕后处理的步骤即可。这种方法的自由性很高，我们可以方便地模拟各种雾效，例如均匀的雾效、基于距离的线性/指数雾效、基于高度的雾效等。在学习完本节后，我们可以得到类似图13.5中的效果。



▲ 图13.5 左边：原效果。右边：添加全局雾效后的效果

基于屏幕后处理的全局雾效的关键是，根据深度纹理来重建每个像素在世界空间下的位置。尽管在13.2节中，我们在模拟运动模糊时已经实现了这个要求，即构建出当前像素的NDC坐标，再通过当前摄像机的视角\*投影矩阵的逆矩阵来得到世界空间下的像素坐标，但是，这样的实现需要在片元着色器中进行矩阵乘法的操作，而这通常会影响游戏性能。在本节中，我们将会学习一个快速从深度纹理中重建世界坐标的方法。这种方法首先对图像空间下的视锥体射线（从摄像机出发，指向图像上的某点的射线）进行插值，这条射线存储了该像素在世界空间下到摄像机的方向信息。然后，我们把该射线和线性化后的

视角空间下的深度值相乘，再加上摄像机的世界位置，就可以得到该像素在世界空间下的位置。当我们得到世界坐标后，就可以轻松地使用各个公式来模拟全局雾效了。

### 13.3.1 重建世界坐标

在开始动手写代码之前，我们首先来了解如何从深度纹理中重建世界坐标。我们知道，坐标系中的一个顶点坐标可以通过它相对于另一个顶点坐标的偏移量来求得。重建像素的世界坐标也是基于这样的思想。我们只需要知道摄像机在世界空间下的位置，以及世界空间下该像素相对于摄像机的偏移量，把它们相加就可以得到该像素的世界坐标。整个过程可以使用下面的代码来表示：

```
float4 worldPos = _WorldSpaceCameraPos + linearDepth *  
interpolatedRay;
```

其中，`_WorldSpaceCameraPos`是摄像机在世界空间下的位置，这可以由Unity的内置变量直接访问得到。而`linearDepth * interpolatedRay`则可以计算得到该像素相对于摄像机的偏移量，`linearDepth`是由深度纹理得到的线性深度值，`interpolatedRay`是由顶点着色器输出并插值后得到的射线，它不仅包含了该像素到摄像机的方向，也包含了距离信息。`linearDepth`的获取我们已经在13.1.2节中详细解释过了，因此，本节着重解释`interpolatedRay`的求法。

`interpolatedRay`来源于对近裁剪平面的4个角的某个特定向量的插值，这4个向量包含了它们到摄像机的方向和距离信息，我们可以利用摄像机的近裁剪平面距离、FOV、纵横比计算而得。图13.6显示了计算时使用的一些辅助向量。为了方便计算，我们可以先计算两个向量—

toTop和toRight，它们是起点位于近裁剪平面中心、分别指向摄像机正上方和正右方的向量。它们的计算公式如下：

$$halfHeight = Near \tan \left( \frac{FOV}{2} \right)$$

$$toTop = camera.up \times halfHeight$$

$$toRight = camera.right \times halfHeight \cdot aspect$$

其中，Near是近裁剪平面的距离，FOV是竖直方向的视角范围，camera.up、camera.right分别对应了摄像机的正上方和正右方。

当得到这两个辅助向量后，我们就可以计算4个角相对于摄像机的方向了。我们以左上角为例（见图13.6中的TL点），它的计算公式如下：

$$TL = camera.forward \cdot Near + toTop - toRight$$

读者可以依靠基本的矢量运算验证上面的结果。同理，其他3个角的计算也是类似的：

$$TR = camera.forward \cdot Near + toTop + toRight$$

$$BL = camera.forward \cdot Near - toTop - toRight$$

$$BR = camera.forward \cdot Near - toTop + toRight$$

注意，上面求得的4个向量不仅包含了方向信息，它们的模对应了4个点到摄像机的空间距离。由于我们得到的线性深度值并非是点到摄

像机的欧式距离，而是在 $z$ 方向上的距离，因此，我们不能直接使用深度值和4个角的单位方向的乘积来计算它们到摄像机的偏移量，如图13.7所示。想要把深度值转换成到摄像机的欧式距离也很简单，我们以TL点为例，根据相似三角形原理，TL所在的射线上，像素的深度值和它到摄像机的实际距离的比等于近裁剪平面的距离和TL向量的模的比，即

$$\frac{depth}{dist} = \frac{Near}{|TL|}$$

由此可得，我们需要的TL距离摄像机的欧氏距离 $dist$ :

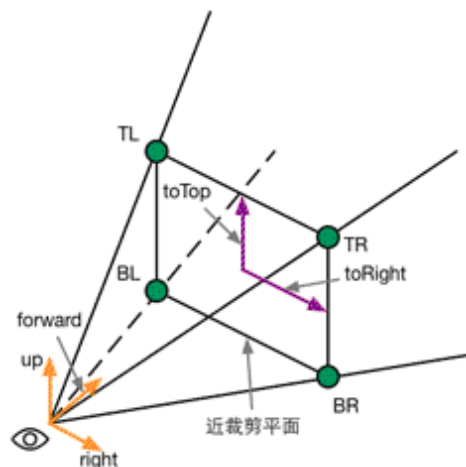
$$dist = \frac{|TL|}{Near} \times depth$$

由于4个点相互对称，因此其他3个向量的模和TL相等，即我们可以使用同一个因子和单位向量相乘，得到它们对应的向量值：

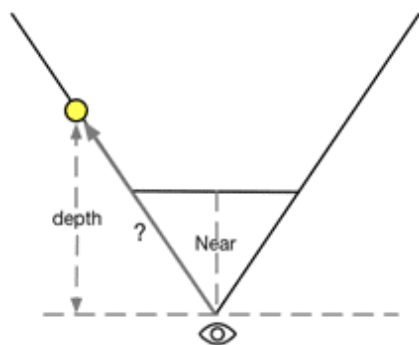
$$scale = \frac{|TL|}{|Near|}$$

$$Ray_{TL} = \frac{TL}{|TL|} \times scale, Ray_{TR} = \frac{TR}{|TR|} \times scale$$

$$Ray_{BL} = \frac{BL}{|BL|} \times scale, Ray_{BR} = \frac{BR}{|BR|} \times scale$$



▲ 图13.6 计算interpolatedRay



▲ 图13.7 采样得到的深度值并非是点到摄像机的欧式距离

屏幕后处理的原理是使用特定的材质去渲染一个刚好填充整个屏幕的四边形面片。这个四边形面片的4个顶点就对应了近裁剪平面的4个角。因此，我们可以把上面的计算结果传递给顶点着色器，顶点着色器根据当前的位置选择它所对应的向量，然后再将其输出，经插值后传递给片元着色器得到interpolatedRay，我们就可以直接利用本节一开始提到的公式重建该像素在世界空间下的位置了。

### 13.3.2 雾的计算



在简单的雾效实现中，我们需要计算一个雾效系数 $f$ ，作为混合原始颜色和雾的颜色的混合系数：

```
float3 afterFog = f * fogColor + (1 - f) * origColor;
```

这个雾效系数 $f$ 有很多计算方法。在Unity内置的雾效实现中，支持三种雾的计算方式—线性（Linear）、指数（Exponential）以及指数的平方（Exponential Squared）。当给定距离 $z$ 后， $f$ 的计算公式分别如下：

Linear:

$$f = \frac{d_{max} - |z|}{d_{max} - d_{min}}$$
 $d_{min}$ 和 $d_{max}$ 分别表示受雾影响的最小距离和最大距离。

Exponential:

$f = [e^{-d \cdot |z|}]$ ， $d$ 是控制雾的浓度的参数。

Exponential Squared:

$f = e^{-(d \cdot |z|)^2}$ ， $d$ 是控制雾的浓度的参数。

在本节中，我们将使用类似线性雾的计算方式，计算基于高度的雾效。具体方法是，当给定一点在世界空间下的高度 $y$ 后， $f$ 的计算公式为：

$$f = \frac{H_{end} - y}{H_{end} - H_{start}}$$
 $H_{start}$ 和 $H_{end}$ 分别表示受雾影响的起始高度和终止高度。

### 13.3.3 实现

为了在Unity中实现基于屏幕后处理的雾效，我们需要进行如下准备工作。

(1) 新建一个场景。在本书资源中，该场景名为Scene\_13\_3。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window -> Lighting -> Skybox中去掉场景中的天空盒子。

(2) 我们需要搭建一个测试雾效的场景。在本书资源的实现中，我们构建了一个包含3面墙的房间，并放置了两个立方体和两个球体，它们都使用了我们在9.5节中创建的标准材质。同时，我们把本书资源中的Translating.cs脚本拖曳给摄像机，让其在场景中不断运动。

(3) 新建一个脚本。在本书资源中，该脚本名为FogWithDepthTexture.cs。把该脚本拖曳到摄像机上。

(4) 新建一个Unity Shader。在本书资源中，该Shader名为Chapter13-FogWithDepthTexture。

我们首先来编写FogWithDepthTexture.cs脚本。打开该脚本，并进行如下修改。

(1) 首先，继承12.1节中创建的基类：

```
public class FogWithDepthTexture : PostEffectsBase {
```

(2) 声明该效果需要的Shader，并据此创建相应的材质：

```
public Shader fogShader;
private Material fogMaterial = null;
public Material material {
    get {
        fogMaterial = CheckShaderAndCreateMaterial(fogShader,
fogMaterial);
        return fogMaterial;
    }
}
```

(3) 在本节中，我们需要获取摄像机的相关参数，如近裁剪平面的距离、FOV等，同时还需要获取摄像机在世界空间下的前方、上方和右方等方向，因此我们用两个变量存储摄像机的Camera组件和Transform组件：

```
private Camera myCamera;
public Camera camera {
    get {
        if (myCamera == null) {
            myCamera = GetComponent<Camera>();
        }
        return myCamera;
    }
}
private Transform myCameraTransform;
public Transform cameraTransform {
    get {
        if (myCameraTransform == null) {
            myCameraTransform = camera.transform;
        }
        return myCameraTransform;
    }
}
```

(4) 定义模拟雾效时使用的各个参数：

```
[Range(0.0f, 3.0f)]
public float fogDensity = 1.0f;
public Color fogColor = Color.white;
public float fogStart = 0.0f;
public float fogEnd = 2.0f;
```

`fogDensity`用于控制雾的浓度，`fogColor`用于控制雾的颜色。我们使用的雾效模拟函数是基于高度的，因此参数`fogStart`用于控制雾效的起始高度，`fogEnd`用于控制雾效的终止高度。

(5) 由于本例需要获取摄像机的深度纹理，我们在脚本的`OnEnable`函数中设置摄像机的相应状态：

```
void OnEnable() {  
    camera.depthTextureMode |= DepthTextureMode.Depth;  
}
```

(6) 最后，我们实现了`OnRenderImage`函数：

```
void OnRenderImage (RenderTexture src, RenderTexture dest) {  
    if (material != null) {  
        Matrix4x4 frustumCorners = Matrix4x4.identity;  
        float fov = camera.fieldOfView;  
        float near = camera.nearClipPlane;  
        float far = camera.farClipPlane;  
        float aspect = camera.aspect;  
        float halfHeight = near * Mathf.Tan(fov * 0.5f *  
Mathf.Deg2Rad);  
        Vector3 toRight = cameraTransform.right * halfHeight *  
aspect;  
        Vector3 toTop = cameraTransform.up * halfHeight;  
        Vector3 topLeft = cameraTransform.forward * near + toTop -  
toRight;  
        float scale = topLeft.magnitude / near;  
        topLeft.Normalize();  
        topLeft *= scale;  
        Vector3 topRight = cameraTransform.forward * near + toRight  
+ toTop;  
        topRight.Normalize();  
        topRight *= scale;  
        Vector3 bottomLeft = cameraTransform.forward * near - toTop  
- toRight;  
        bottomLeft.Normalize();  
        bottomLeft *= scale;  
        Vector3 bottomRight = cameraTransform.forward * near +  
toRight - toTop;  
        bottomRight.Normalize();  
        bottomRight *= scale;  
        frustumCorners.SetRow(0, bottomLeft);
```

```

        frustumCorners.SetRow(1, bottomRight);
        frustumCorners.SetRow(2, topRight);
        frustumCorners.SetRow(3, topLeft);
        material.SetMatrix("_FrustumCornersRay", frustumCorners);
        material.SetMatrix("_ViewProjectionInverseMatrix",
(camera.projectionMatrix *
camera.worldToCameraMatrix).inverse);
        material.SetFloat("_FogDensity", fogDensity);
        material.SetColor("_FogColor", fogColor);
        material.SetFloat("_FogStart", fogStart);
        material.SetFloat("_FogEnd", fogEnd);
        Graphics.Blit (src, dest, material);
    } else {
        Graphics.Blit(src, dest);
    }
}

```

**OnRenderImage**首先计算了近裁剪平面的四个角对应的向量，并把它们存储在一个矩阵类型的变量（**frustumCorners**）中。计算过程我们已经在13.3.1节中详细解释过了，代码只是套用了之前讲过的公式而已。我们按一定顺序把这四个方向存储到了**frustumCorners**不同的行中，这个顺序是非常重要的，因为这决定了我们在顶点着色器中使用哪一行作为该点的待插值向量。随后，我们把结果和其他参数传递给材质，并调用**Graphics.Blit (src, dest, material)**把渲染结果显示在屏幕上。

下面，我们来实现**Shader**的部分。打开**Chapter13-FogWithDepthTexture**，进行如下修改。

(1) 我们首先需要声明本例使用的各个属性：

```

Properties {
    _MainTex ("Base (RGB)", 2D) = "white" {}
    _FogDensity ("Fog Density", Float) = 1.0
    _FogColor ("Fog Color", Color) = (1, 1, 1, 1)
    _FogStart ("Fog Start", Float) = 0.0
    _FogEnd ("Fog End", Float) = 1.0
}

```

(2) 在本节中，我们使用CGINCLUDE来组织代码。我们在SubShader块中利用CGINCLUDE和ENDCG语义来定义一系列代码：

```
SubShader {  
    CGINCLUDE  
    ...  
    ENDCG  
    ...  
}
```

(3) 声明代码中需要使用的各个变量：

```
float4x4 _FrustumCornersRay;  
sampler2D _MainTex;  
half4 _MainTex_TexelSize;  
sampler2D _CameraDepthTexture;  
half _FogDensity;  
fixed4 _FogColor;  
float _FogStart;  
float _FogEnd;
```

\_FrustumCornersRay虽然没有在Properties中声明，但仍可由脚本传递给Shader。除了Properties中声明的各个属性，我们还声明了深度纹理\_CameraDepthTexture，Unity会在背后把得到的深度纹理传递给该值。

(4) 定义顶点着色器：

```
struct v2f {  
    float4 pos : SV_POSITION;  
    half2 uv : TEXCOORD0;  
    half2 uv_depth : TEXCOORD1;  
    float4 interpolatedRay : TEXCOORD2;  
};  
v2f vert(appdata_img v) {  
    v2f o;  
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);  
    o.uv = v.texcoord;  
    o.uv_depth = v.texcoord;  
    #if UNITY_UV_STARTS_AT_TOP  
    if (_MainTex_TexelSize.y < 0)  
        o.uv_depth.y = 1 - o.uv_depth.y;  
    #endif  
}
```

```

int index = 0;
if (v.texcoord.x < 0.5 && v.texcoord.y < 0.5) {
    index = 0;
} else if (v.texcoord.x > 0.5 && v.texcoord.y < 0.5) {
    index = 1;
} else if (v.texcoord.x > 0.5 && v.texcoord.y > 0.5) {
    index = 2;
} else {
    index = 3;
}
#ifdef UNITY_UV_STARTS_AT_TOP
    if (_MainTex_TexelSize.y < 0)
        index = 3 - index;
#endif
o.interpolatedRay = _FrustumCornersRay[index];
return o;
}

```

在**v2f**结构体中，我们除了定义顶点位置、屏幕图像和深度纹理的纹理坐标外，还定义了**interpolatedRay**变量存储插值后的像素向量。在顶点着色器中，我们对深度纹理的采样坐标进行了平台差异化处理。更重要的是，我们要决定该点对应了4个角中的哪个角。我们采用的方法是判断它的纹理坐标。我们知道，在**Unity**中，纹理坐标的(0, 0)点对应了左下角，而(1, 1)点对应了右上角。我们据此来判断该顶点对应的索引，这个对应关系和我们在脚本中对**frustumCorners**的赋值顺序是一致的。实际上，不同平台的纹理坐标不一定是满足上面的条件的，例如**DirectX**和**Metal**这样的平台，左上角对应了(0, 0)点，但大多数情况下**Unity**会把这些平台下的屏幕图像进行翻转，因此我们仍然可以利用这个条件。但如果在类似**DirectX**的平台上开启了抗锯齿，**Unity**就不会进行这个翻转。为了此时仍然可以得到相应顶点位置的索引值，我们对索引值也进行了平台差异化处理（详见5.6.1节），以便在必要时也对索引值进行翻转。最后，我们使用索引值来获取**\_FrustumCornersRay**中对应的行作为该顶点的**interpolatedRay**值。



尽管我们这里使用了很多判断语句，但由于屏幕后处理所用的模型是一个四边形网格，只包含4个顶点，因此这些操作不会对性能造成很大影响。

(5) 我们定义了片元着色器来产生雾效：

```
fixed4 frag(v2f i) : SV_Target {  
    float linearDepth =  
LinearEyeDepth(SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, i.  
uv_depth));  
    float3 worldPos = _WorldSpaceCameraPos + linearDepth *  
i.interpolatedRay.xyz;  
    float fogDensity = (_FogEnd - worldPos.y) / (_FogEnd -  
_FogStart);  
    fogDensity = saturate(fogDensity * _FogDensity);  
    fixed4 finalColor = tex2D(_MainTex, i.uv);  
    finalColor.rgb = lerp(finalColor.rgb, _FogColor.rgb,  
fogDensity);  
    return finalColor;  
}
```

首先，我们需要重建该像素在世界空间中的位置。为此，我们首先使用SAMPLE\_DEPTH\_TEXTURE对深度纹理进行采样，再使用LinearEyeDepth得到视角空间下的线性深度值。之后，与interpolatedRay相乘后再和世界空间下的摄像机位置相加，即可得到世界空间下的位置。

得到世界坐标后，模拟雾效就变得非常容易。在本例中，我们选择实现基于高度的雾效模拟，计算公式可参见13.3.2节。我们根据材质属性\_FogEnd和\_FogStart计算当前的像素高度worldPos.y对应的雾效系数fogDensity，再和参数\_FogDensity相乘后，利用saturate函数截取到[0, 1]范围内，作为最后的雾效系数。然后，我们使用该系数将雾的颜色和原始颜色进行混合后返回。读者也可以使用不同的公式来实现其他种类的雾效。

(6) 随后，我们定义了雾效渲染所需的Pass:

```
Pass {  
    ZTest Always Cull Off ZWrite Off  
    CGPROGRAM  
    #pragma vertex vert  
    #pragma fragment frag  
    ENDCG  
}
```

(7) 最后，我们关闭了Shader的Fallback:

```
Fallback Off
```

完成后返回编辑器，并把Chapter13-FogWithDepthTexture拖曳到摄像机的FogWithDepthTexture.cs脚本中的fogShader参数中。当然，我们可以在FogWithDepthTexture.cs的脚本面板中将fogShader参数的默认值设置为Chapter13-FogWithDepthTexture，这样就不需要以后使用时每次都手动拖曳了。

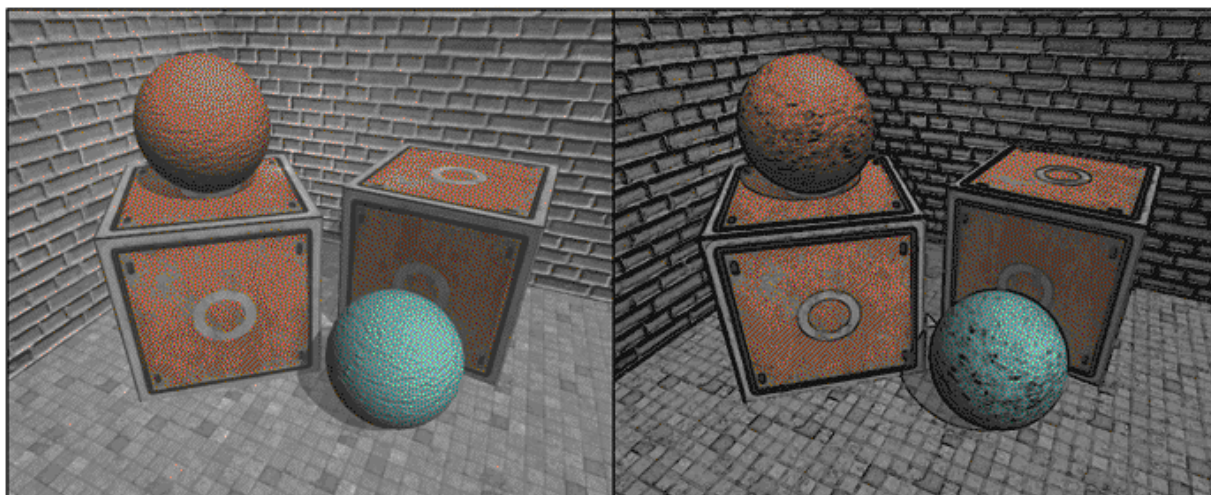
本节介绍的使用深度纹理重建像素的世界坐标的方法是非常有用的。但需要注意的是，这里的实现是基于摄像机的投影类型是透视投影的前提下。如果需要在正交投影的情况下重建世界坐标，需要使用不同的公式，但请读者相信，这个过程不会比透视投影的情况更加复杂。有兴趣的读者可以尝试自行推导，或参考这篇博客

(<http://www.derschmale.com/2014/03/19/reconstructing-positions-from-the-depth-buffer-pt-2-perspective-and-orthographic-general-case/>) 来实现。

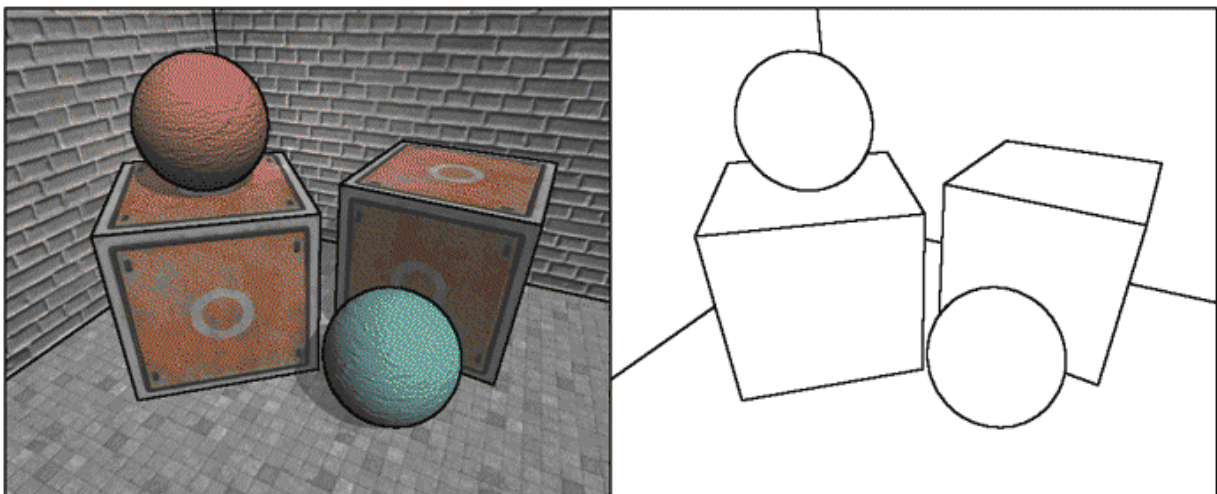
## 13.4 再谈边缘检测

在12.3节中，我们曾介绍如何使用Sobel算子对屏幕图像进行边缘检测，实现描边的效果。但是，这种直接利用颜色信息进行边缘检测的方法会产生很多我们不希望得到的边缘线，如图13.8所示。

可以看出，物体的纹理、阴影等位置也被描上黑边，而这往往不是我们希望看到的。在本节中，我们将学习如何在深度和法线纹理上进行边缘检测，这些图像不会受纹理和光照的影响，而仅仅保存了当前渲染物体的模型信息，通过这样的方式检测出来的边缘更加可靠。在学习完本节后，我们可以得到类似图13.9中的效果。



▲图13.8 左边：原效果，右边：直接对颜色图像进行边缘检测的结果



▲ 图13.9 在深度和法线纹理上进行更健壮的边缘检测。左边：在原图上描边的效果。右边：只显示描边的效果

与12.3节使用Sobel算子不同，本节将使用Roberts算子来进行边缘检测。它使用的卷积核如图13.10所示。

**Roberts**

-1	0
0	1

$G_x$

0	-1
1	0

$G_y$

▲ 图13.10 Roberts算子

Roberts算子的本质就是计算左上角和右下角的差值，乘以右上角和左下角的差值，作为评估边缘的依据。在下面的实现中，我们也会

按这样的方式，取对角方向的深度或法线值，比较它们之间的差值，如果超过某个阈值（可由参数控制），就认为它们之间存在一条边。

首先，我们需要进行如下准备工作。

(1) 新建一个场景。在本书资源中，该场景名为**Scene\_13\_4**。在**Unity 5.2**中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在**Window → Lighting → Skybox**中去掉场景中的天空盒子。

(2) 我们需要搭建一个测试雾效的场景。在本书资源的实现中，我们构建了一个包含3面墙的房间，并放置了两个立方体和两个球体，它们都使用了我们在9.5节中创建的标准材质。同时，我们把本书资源中的**Translating.cs**脚本拖曳给摄像机，让其在场景中不断运动。

(3) 新建一个脚本。在本书资源中，该脚本名为**EdgeDetectNormalsAndDepth.cs**。把该脚本拖曳到摄像机上。

(4) 新建一个**Unity Shader**。在本书资源中，该**Shader**名为**Chapter13-EdgeDetectNormalAndDepth**。

我们首先来编写**EdgeDetectNormalsAndDepth.cs**脚本。该脚本与12.3节中实现的**EdgeDetection.cs**脚本几乎完全一样，只是添加了一些新的属性。为了完整性，我们再次说明对该脚本进行的修改。

(1) 首先，继承12.1节中创建的基类：

```
public class EdgeDetectNormalsAndDepth : PostEffectsBase {
```

(2) 声明该效果需要的Shader，并据此创建相应的材质：

```
public Shader edgeDetectShader;
private Material edgeDetectMaterial = null;
public Material material {
    get {
        edgeDetectMaterial =
        CheckShaderAndCreateMaterial(edgeDetectShader, edgeDetectMaterial);
        return edgeDetectMaterial;
    }
}
```

(3) 在脚本中提供了调整边缘线强度描边颜色以及背景颜色的参数。同时添加了控制采样距离以及对深度和法线进行边缘检测时的灵敏度参数：

```
[Range(0.0f, 1.0f)]
public float edgesOnly = 0.0f;
public Color edgeColor = Color.black;
public Color backgroundColor = Color.white;
public float sampleDistance = 1.0f;
public float sensitivityDepth = 1.0f;
public float sensitivityNormals = 1.0f;
```

**sampleDistance**用于控制对深度+法线纹理采样时，使用的采样距离。从视觉上来看，**sampleDistance**值越大，描边越宽。

**sensitivityDepth**和**sensitivityNormals**将会影响当邻域的深度值或法线值相差多少时，会被认为存在一条边界。如果把灵敏度调得很大，那么可能即使是深度或法线上很小的变化也会形成一条边。

(4) 由于本例需要获取摄像机的深度+法线纹理，我们在脚本的**OnEnable**函数中设置摄像机的相应状态：

```
void OnEnable() {
    GetComponent<Camera>().depthTextureMode |=
    DepthTextureMode.DepthNormals;
}
```

(5) 实现OnRenderImage函数，把各个参数传递给材质：

```
[ImageEffectOpaque]
void OnRenderImage (RenderTexture src, RenderTexture dest) {
    if (material != null) {
        material.SetFloat("_EdgeOnly", edgesOnly);
        material.SetColor("_EdgeColor", edgeColor);
        material.SetColor("_BackgroundColor", backgroundColor);
        material.SetFloat("_SampleDistance", sampleDistance);
        material.SetVector("_Sensitivity", new
Vector4(sensitivityNormals, sensitivityDepth, 0.0f, 0.0f));
        Graphics.Blit(src, dest, material);
    } else {
        Graphics.Blit(src, dest);
    }
}
```

需要注意的是，这里我们为OnRenderImage函数添加了[ImageEffectOpaque]属性。我们曾在12.1节中提到过该属性的含义。在默认情况下，OnRenderImage函数会在所有的不透明和透明的Pass执行完毕后被调用，以便对场景中所有游戏对象都产生影响。但有时，我们希望在透明的Pass（即渲染队列小于等于2 500的Pass，内置的Background、Geometry和AlphaTest渲染队列均在此范围内）执行完毕后立即调用该函数，而不对透明物体（渲染队列为Transparent的Pass）产生影响，此时，我们可以在OnRenderImage函数前添加ImageEffectOpaque属性来实现这样的目的。在本例中，我们只希望对不透明物体进行描边，而不希望透明物体也被描边，因此需要添加该属性。

下面，我们来实现Shader的部分。打开Chapter13-EdgeDetectNormalAndDepth，进行如下修改。

(1) 我们首先需要声明本例使用的各个属性：



```
Properties {  
    _MainTex ("Base (RGB)", 2D) = "white" {}  
    _EdgeOnly ("Edge Only", Float) = 1.0  
    _EdgeColor ("Edge Color", Color) = (0, 0, 0, 1)  
    _BackgroundColor ("Background Color", Color) = (1, 1, 1, 1)  
    _SampleDistance ("Sample Distance", Float) = 1.0  
    _Sensitivity ("Sensitivity", Vector) = (1, 1, 1, 1)  
}
```

其中，`_Sensitivity`的xy分量分别对应了法线和深度的检测灵敏度，zw分量则没有实际用途。

(2) 在本节中，我们使用**CGINCLUDE**来组织代码。我们在SubShader块中利用**CGINCLUDE**和**ENDCG**语义来定义一系列代码：

```
SubShader {  
    CGINCLUDE  
    ...  
    ENDCG  
    ...  
}
```

(3) 为了在代码中访问各个属性，我们需要在CG代码块中声明对应的变量：

```
sampler2D _MainTex;  
half4 _MainTex_TexelSize;  
fixed _EdgeOnly;  
fixed4 _EdgeColor;  
fixed4 _BackgroundColor;  
float _SampleDistance;  
half4 _Sensitivity;  
sampler2D _CameraDepthNormalsTexture;
```

在上面的代码中，我们声明了需要获取的深度+法线纹理 `_CameraDepthNormalsTexture`。由于我们需要对邻域像素进行纹理采样，所以还声明了存储纹素大小的变量 `_MainTex_TexelSize`。

(4) 定义顶点着色器：

```

struct v2f {
    float4 pos : SV_POSITION;
    half2 uv[5]: TEXCOORD0;
};
v2f vert(appdata_img v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);
    half2 uv = v.texcoord;
    o.uv[0] = uv;
    #if UNITY_UV_STARTS_AT_TOP
    if (_MainTex_TexelSize.y < 0)
        uv.y = 1 - uv.y;
    #endif
    o.uv[1] = uv + _MainTex_TexelSize.xy * half2(1,1) *
_SampleDistance;
    o.uv[2] = uv + _MainTex_TexelSize.xy * half2(-1,-1) *
_SampleDistance;
    o.uv[3] = uv + _MainTex_TexelSize.xy * half2(-1,1) *
_SampleDistance;
    o.uv[4] = uv + _MainTex_TexelSize.xy * half2(1,-1) *
_SampleDistance;
    return o;
}

```

我们在v2f结构体中定义了一个维数为5的纹理坐标数组。这个数组的第一个坐标存储了屏幕颜色图像的采样纹理。我们对深度纹理的采样坐标进行了平台差异化处理，在必要时对它的竖直方向进行了翻转。数组中剩余的4个坐标则存储了使用Roberts算子时需要采样的纹理坐标，我们还使用了\_SampleDistance来控制采样距离。通过把计算采样纹理坐标的代码从片元着色器中转移到顶点着色器中，可以减少运算，提高性能。由于从顶点着色器到片元着色器的插值是线性的，因此这样的转移并不会影响纹理坐标的计算结果。

(5) 然后，我们定义了片元着色器：

```

fixed4 fragRobertsCrossDepthAndNormal(v2f i) : SV_Target {
    half4 sample1 = tex2D(_CameraDepthNormalsTexture, i.uv[1]);
    half4 sample2 = tex2D(_CameraDepthNormalsTexture, i.uv[2]);
    half4 sample3 = tex2D(_CameraDepthNormalsTexture, i.uv[3]);
    half4 sample4 = tex2D(_CameraDepthNormalsTexture, i.uv[4]);
}

```

```

    half edge = 1.0;
    edge *= CheckSame(sample1, sample2);
    edge *= CheckSame(sample3, sample4);
    fixed4 withEdgeColor = lerp(_EdgeColor, tex2D(_MainTex,
i.uv[0]), edge);
    fixed4 onlyEdgeColor = lerp(_EdgeColor, _BackgroundColor,
edge);
    return lerp(withEdgeColor, onlyEdgeColor, _EdgeOnly);
}

```

我们首先使用4个纹理坐标对深度+法线纹理进行采样，再调用CheckSame函数来分别计算对角线上两个纹理值的差值。CheckSame函数的返回值要么是0，要么是1，返回0时表明这两点之间存在一条边界，反之则返回1。它的定义如下：

```

half CheckSame(half4 center, half4 sample) {
    half2 centerNormal = center.xy;
    float centerDepth = DecodeFloatRG(center.zw);
    half2 sampleNormal = sample.xy;
    float sampleDepth = DecodeFloatRG(sample.zw);
    // difference in normals
    // do not bother decoding normals - there's no need here
    half2 diffNormal = abs(centerNormal - sampleNormal) *
_Sensitivity.x;
    int isSameNormal = (diffNormal.x + diffNormal.y) < 0.1;
    // difference in depth
    float diffDepth = abs(centerDepth - sampleDepth) *
_Sensitivity.y;
    // scale the required threshold by the distance
    int isSameDepth = diffDepth < 0.1 * centerDepth;
    // return:
    // 1 - if normals and depth are similar enough
    // 0 - otherwise
    return isSameNormal * isSameDepth ? 1.0 : 0.0;
}

```

CheckSame首先对输入参数进行处理，得到两个采样点的法线和深度值。值得注意的是，这里我们并没有解码得到真正的法线值，而是直接使用了xy分量。这是因为我们只需要比较两个采样值之间的差异度，而并不需要知道它们真正的法线值。然后，我们把两个采样点的

对应值相减并取绝对值，再乘以灵敏度参数，把差异值的每个分量相加再和一个阈值比较，如果它们的和小于阈值，则返回1，说明差异不明显，不存在一条边界；否则返回0。最后，我们把法线和深度的检查结果相乘，作为组合后的返回值。

当通过CheckSame函数得到边缘信息后，片元着色器就利用该值进行颜色混合，这和12.3节中的步骤一致。

(6) 然后，我们定义了边缘检测需要使用的Pass:

```
Pass {  
    ZTest Always Cull Off ZWrite Off  
    CGPROGRAM  
    #pragma vertex vert  
    #pragma fragment fragRobertsCrossDepthAndNormal  
    ENDCG  
}
```

(7) 最后，我们关闭了该Shader的Fallback:

```
Fallback Off
```

完成后返回编辑器，并把Chapter13-EdgeDetectNormalAndDepth拖曳到摄像机的EdgeDetect NormalsAndDepth.cs脚本中的edgeDetectShader参数中。当然，我们可以在EdgeDetectNormals AndDepth.cs的脚本面板中将edgeDetectShader参数的默认值设置为Chapter13-EdgeDetectNormal AndDepth，这样就不需要以后使用时每次都手动拖曳了。

本节实现的描边效果是基于整个屏幕空间进行的，也就是说，场景内的所有物体都会被添加描边效果。但有时，我们希望只对特定的物体进行描边，例如当玩家选中场景中的某个物体后，我们想要在该

物体周围添加一层描边效果。这时，我们可以使用Unity提供的Graphics.DrawMesh或Graphics.DrawMeshNow函数把需要描边的物体再次渲染一遍（在所有不透明物体渲染完毕之后），然后再使用本节提到的边缘检测算法计算深度或法线纹理中每个像素的梯度值，判断它们是否小于某个阈值，如果是，就在Shader中使用clip()函数将该像素剔除掉，从而显示出原来的物体颜色。

## 13.5 扩展阅读

在本章中，我们介绍了如何使用深度和法线纹理实现诸如全局雾效、边缘检测等效果。尽管我们只使用了深度和法线纹理，但实际上我们可以在Unity中创建任何需要的缓存纹理。这可以通过使用Unity的着色器替换（Shader Replacement）功能（即调用Camera.RenderWithShader(shader, replacementTag)函数）把整个场景再次渲染一遍来得到，而在很多时候，这实际也是Unity创建深度和法线纹理时使用的方法。

深度和法线纹理在屏幕特效的实现中往往扮演了重要的角色。许多特殊的屏幕效果都需要依靠这两种纹理的帮助。Unity曾在2011年的SIGGRAPH（计算机图形学的顶级会议）上做了一个关于使用深度纹理实现各种特效的演讲（<http://blogs.unity3d.com/2011/09/08/special-effects-with-depth-talk-at-siggraph/>）。在这个演讲中，Unity的工作人员解释了如何利用深度纹理来实现特定物体的描边、角色护盾、相交线的高光模拟等效果。在Unity的Image Effect（<http://docs.unity3d.com/Manual/comp-ImageEffects.html>）包中，读者也可以找到一些传统的使

用深度纹理实现屏幕特效的例子，例如屏幕空间的环境遮挡（Screen Space Ambient Occlusion, SSAO）等效果。

## 第14章 非真实感渲染

尽管游戏渲染一般都是以**照相写实主义（photorealism）**作为主要目标，但也有许多游戏使用了**非真实感渲染（Non-Photorealistic Rendering, NPR）**的方法来渲染游戏画面。非真实感渲染的一个主要目标是，使用一些渲染方法使得画面达到和某些特殊的绘画风格相似的效果，例如卡通、水彩风格等。

在本章中，我们将会介绍两种常见的非真实感渲染方法。在14.1节中，我们将会学习如何实现一个包含了简单漫反射、高光和描边的卡通风格的渲染效果。14.2节将会介绍一种实时素描效果的实现。在本章最后，我们还会给出一些关于非真实感渲染的资料，读者可以在这些文献中找到更多非真实感渲染的实现方法。

### 14.1 卡通风格的渲染

卡通风格是游戏中常见的一种渲染风格。使用这种风格的游戏画面通常有一些共有的特点，例如物体都被黑色的线条描边，以及分明的明暗变化等。由日本卡普空（英文名：Capcom）株式会社开发的游戏《大神》（英文名：Okami）就使用了水墨+卡通风格来渲染整个画面，如图14.1所示，这种渲染风格获得了广泛赞誉。

要实现卡通渲染有很多方法，其中之一就是使用**基于色调的着色技术（tone-based shading）**。Gooch等人在他们1998年的一篇论文<sup>[1]</sup>中



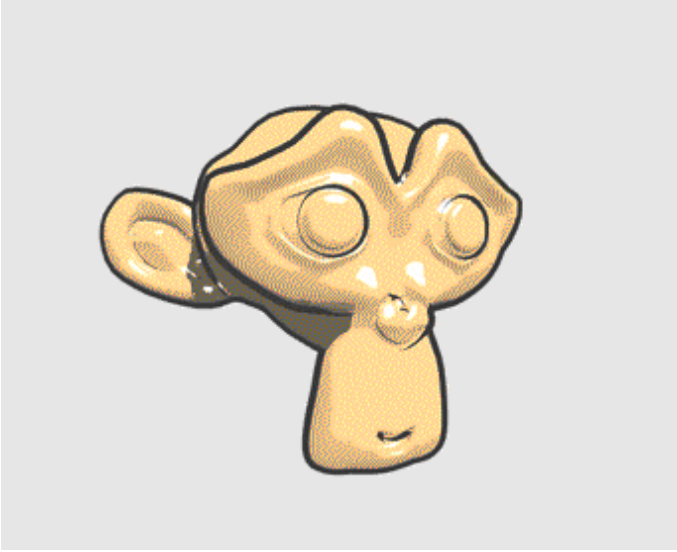
提出并实现了基于色调的光照模型。在实现中，我们往往会使用漫反射系数对一张一维纹理进行采样，以控制漫反射的色调。我们曾在7.3节使用渐变纹理实现过这样的效果。卡通风格的高光效果也和我们之前学习的光照不同。在卡通风格中，模型的高光往往是一块块分界明显的纯色区域。

除了光照模型不同外，卡通风格通常还需要在物体边缘部分绘制轮廓。在之前的章节中，我们曾介绍使用屏幕后处理技术对屏幕图像进行描边。在本节，我们将会介绍基于模型的描边方法，这种方法的实现更加简单，而且在很多情况下也能得到不错的效果。

在本节结束后，我们将会实现类似图14.2的效果。



▲ 图14.1 游戏《大神》（英文名：Okami）的游戏截图



▲图14.2 卡通风格的渲染效果

### 14.1.1 渲染轮廓线

在实时渲染中，轮廓线的渲染是应用非常广泛的一种效果。近20年来，有许多绘制模型轮廓线的方法被先后提出来。在《Real Time Rendering, third edition》一书中，作者把这些方法分成了5种类型。

- 基于观察角度和表面法线的轮廓线渲染。这种方法使用视角方向和表面法线的点乘结果来得到轮廓线的信息。这种方法简单快速，可以在一个Pass中就得到渲染结果，但局限性很大，很多模型渲染出来的描边效果都不尽如人意。
- 过程式几何轮廓线渲染。这种方法的核心是使用两个Pass渲染。第一个Pass渲染背面的面片，并使用某些技术让它的轮廓可见；第二个Pass再正常渲染正面的面片。这种方法的优点在于快速有效，并且适用于绝大多数表面平滑的模型，但它的缺点是不适合类似于立方体这样平整的模型。

- 基于图像处理的轮廓线渲染。我们在第12、13章介绍的边缘检测的方法就属于这个类别。这种方法的优点在于，可以适用于任何种类的模型。但它也有自身的局限所在，一些深度和法线变化很小的轮廓无法被检测出来，例如桌子上的纸张。
- 基于轮廓边检测的轮廓线渲染。上面提到的各种方法，一个最大的问题是，无法控制轮廓线的风格渲染。对于一些情况，我们希望可以渲染出独特风格的轮廓线，例如水墨风格等。为此，我们希望可以检测出精确的轮廓边，然后直接渲染它们。检测一条边是否是轮廓边的公式很简单，我们只需要检查和这条边相邻的两个三角面片是否满足以下条件：

$$(\mathbf{n}_0 \cdot \mathbf{v} > 0) \neq (\mathbf{n}_1 \cdot \mathbf{v} > 0)$$

其中， $\mathbf{n}_0$ 和 $\mathbf{n}_1$ 分别表示两个相邻三角面片的法向， $\mathbf{v}$ 是从视角到该边上任意顶点的方向。上述公式的本质在于检查两个相邻的三角面片是否一个朝正面、一个朝背面。我们可以在几何着色器（Geometry Shader）的帮助下实现上面的检测过程。当然，这种方法也有缺点，除了实现相对复杂外，它还会有动画连贯性的问题。也就是说，由于是逐帧单独提取轮廓，所以在帧与帧之间会出现跳跃性。

- 最后一个种类就是混合了上述的几种渲染方法。例如，首先找到精确的轮廓边，把模型和轮廓边渲染到纹理中，再使用图像处理的方法识别出轮廓线，并在图像空间下进行风格化渲染。

在本节中，我们将会使用Unity中使用过程式几何轮廓线渲染的方法对模型进行轮廓描边。我们将使用两个Pass渲染模型：在第一个Pass中，我们会使用轮廓线颜色渲染整个背面的面片，并在视角空间下把

模型顶点沿着法线方向向外扩张一段距离，以此来让背部轮廓线可见。代码如下：

```
viewPos = viewPos + viewNormal * _Outline;
```

但是，如果直接使用顶点法线进行扩展，对于一些内凹的模型，就可能发生背面面片遮挡正面面片的情况。为了尽可能防止出现这样的情况，在扩张背面顶点之前，我们首先对顶点法线的 $z$ 分量进行处理，使它们等于一个定值，然后把法线归一化后再对顶点进行扩张。这样的好处在于，扩展后的背面更加扁平化，从而降低了遮挡正面面片的可能性。代码如下：

```
viewNormal.z = -0.5;  
viewNormal = normalize(viewNormal);  
viewPos = viewPos + viewNormal * _Outline;
```

### 14.1.2 添加高光

前面提到过，卡通风格中的高光往往是模型上一块块分界明显的纯色区域。为了实现这种效果，我们就不能再使用之前学习的光照模型。回顾一下，在之前实现Blinn-Phong模型的过程中，我们使用法线点乘光照方向以及视角方向和的一半，再和另一个参数进行指数操作得到高光反射系数。代码如下：

```
float spec = pow(max(0, dot(normal, halfDir)), _Gloss)
```

对于卡通渲染需要的高光反射光照模型，我们同样需要计算`normal`和`halfDir`的点乘结果，但不同的是，我们把该值和一个阈值进行比较，如果小于该阈值，则高光反射系数为0，否则返回1。

```
float spec = dot(worldNormal, worldHalfDir);  
spec = step(threshold, spec);
```

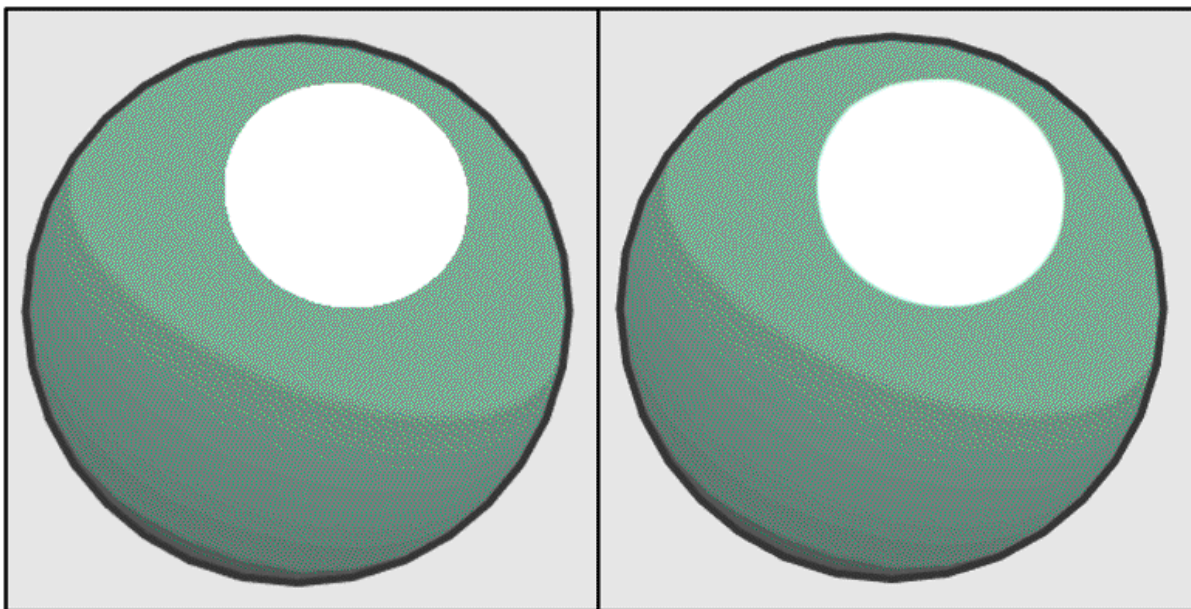
在上面的代码中，我们使用CG的**step函数**来实现和阈值比较的目的。step函数接受两个参数，第一个参数是参考值，第二个参数是待比较的数值。如果第二个参数大于等于第一个参数，则返回1，否则返回0。

但是，这种粗暴的判断方法会在高光区域的边界造成锯齿，如图14.3左图所示。出现这种问题的原因在于，高光区域的边缘不是平滑渐变的，而是由0突变到1。要想对其进行抗锯齿处理，我们可以在边界处很小的一块区域内，进行平滑处理。代码如下：

```
float spec = dot(worldNormal, worldHalfDir);  
spec = lerp(0, 1, smoothstep(-w, w, spec - threshold));
```

在上面的代码中，我们没有像之前一样直接使用step函数返回0或1，而是首先使用了CG的**smoothstep函数**。其中，w是一个很小的值，当spec - threshold小于-w时，返回0，大于w时，返回1，否则在0到1之间进行插值。这样的效果是，我们可以在[-w, w]区间内，即高光区域的边界处，得到一个从0到1平滑变化的spec值，从而实现抗锯齿的目的。尽管我们可以把w设为一个很小的定值，但在本例中，我们选择使用邻域像素之间的近似导数值，这可以通过CG的**fwidth函数**来得到。





▲ 图14.3 左边：未对高光区域进行抗锯齿处理，右边：使用`fwidth`函数对高光区域进行抗锯齿处理

当然，卡通渲染中的高光往往有更多个性化的需要。例如，很多卡通高光特效希望可以随意伸缩、方块化光照区域。**Anjyo**等人在他们2003年的一篇论文<sup>[2]</sup>中给出了一种风格化的卡通高光的实现。读者也可以在这篇非真实感渲染的博文

（<http://blog.csdn.net/candycat1992/article/details/47284289>）中找到这种方法在Unity中的实现。

### 14.1.3 实现

我们现在已经有了理论基础，是时候在Unity中验证我们的结果了。为此，我们需要进行如下准备工作。

(1) 在Unity中新建一个场景。在本书资源中，该场景名为Scene\_14\_1。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个

平行光，并且使用了内置的天空盒子。在Window → Lighting → Skybox中去掉场景中的天空盒子。

(2) 新建一个材质。在本书资源中，该材质名为ToonShadingMat。

(3) 新建一个Unity Shader。在本书资源中，该Unity Shader名为Chapter14-ToonShading。把新的Unity Shader赋给第2步中创建的材质。

(4) 在场景中拖曳一个Suzanne模型，并把第2步中的材质赋给该模型。

(5) 保存场景。

打开Chapter14-ToonShading，关键修改如下。

(1) 首先，我们需要声明本例使用各个属性：

```
Properties {  
    _Color ("Color Tint", Color) = (1, 1, 1, 1)  
    _MainTex ("Main Tex", 2D) = "white" {}  
    _Ramp ("Ramp Texture", 2D) = "white" {}  
    _Outline ("Outline", Range(0, 1)) = 0.1  
    _OutlineColor ("Outline Color", Color) = (0, 0, 0, 1)  
    _Specular ("Specular", Color) = (1, 1, 1, 1)  
    _SpecularScale ("Specular Scale", Range(0, 0.1)) = 0.01  
}
```

其中，\_Ramp是用于控制漫反射色调的渐变纹理，\_Outline用于控制轮廓线宽度，\_OutlineColor对应了轮廓线颜色，\_Specular是高光反射颜色，\_SpecularScale用于控制计算高光反射时使用的阈值。



(2) 定义渲染轮廓线需要的Pass。前面提到过，这个Pass只渲染背面的三角面片，因此，我们需要设置正确的渲染状态：

```
Pass {  
    NAME "OUTLINE"  
  
    Cull Front
```

我们使用Cull指令把正面的三角面片剔除，而只渲染背面。值得注意的是，我们还使用NAME命令为该Pass定义了名称。这是因为，描边在非真实感渲染中是非常常见的效果，为该Pass定义名称可以让我们在后面的使用中不需要再重复编写此Pass，而只需要调用它的名字即可。

(3) 定义描边需要的顶点着色器和片元着色器：

```
v2f vert (a2v v) {  
    v2f o;  
  
    float4 pos = mul(UNITY_MATRIX_MV, v.vertex);  
    float3 normal = mul((float3x3)UNITY_MATRIX_IT_MV, v.normal);  
    normal.z = -0.5;  
    pos = pos + float4(normalize(normal), 0) * _Outline;  
    o.pos = mul(UNITY_MATRIX_P, pos);  
  
    return o;  
}  
  
float4 frag(v2f i) : SV_Target {  
    return float4(_OutlineColor.rgb, 1);  
}
```

如14.1.1节所讲，在顶点着色器中我们首先把顶点和法线变换到视角空间下，这是为了让描边可以在观察空间达到最好的效果。随后，我们设置法线的z分量，对其归一化后再将顶点沿其方向扩张，得到扩张后的顶点坐标。对法线的处理是为了尽可能避免背面扩张后的顶点挡住正面的面片。最后，我们把顶点从视角空间变换到裁剪空间。

片元着色器的代码非常简单，我们只需要用轮廓线颜色渲染整个背面即可。

(4) 然后，我们需要定义光照模型所在的Pass，以渲染模型的正面。由于光照模型需要使用Unity提供的光照等信息，我们需要为Pass进行相应的设置，并添加相应的编译指令：

```
Pass {  
    Tags { "LightMode"="ForwardBase" }  
  
    Cull Back  
  
    CGPROGRAM  
  
    #pragma vertex vert  
    #pragma fragment frag  
  
    #pragma multi_compile_fwdbase
```

在上面的代码中，我们将LightMode设置为ForwardBase，并且使用#pragma语句设置了编译指令，这些都是为了让Shader中的光照变量可以被正确赋值。

(5) 随后，我们定义了顶点着色器：

```
struct v2f {  
    float4 pos : POSITION;  
    float2 uv : TEXCOORD0;  
    float3 worldNormal : TEXCOORD1;  
    float3 worldPos : TEXCOORD2;  
    SHADOW_COORDS(3)  
};  
  
v2f vert (a2v v) {  
    v2f o;  
  
    o.pos = mul( UNITY_MATRIX_MVP, v.vertex);  
    o.uv = TRANSFORM_TEX (v.texcoord, _MainTex);  
    o.worldNormal = mul(v.normal, (float3x3)_World2Object);  
    o.worldPos = mul(_Object2World, v.vertex).xyz;
```

```
TRANSFER_SHADOW(o);  
  
return o;  
}
```

在上面的代码中，我们计算了世界空间下的法线方向和顶点位置，并使用Unity提供的内置宏SHADOW\_COORDS和TRANSFER\_SHADOW来计算阴影所需的各个变量。这些宏的实现原理可以参见9.4节。

(6) 片元着色器中包含了计算光照模型的关键代码：

```
float4 frag(v2f i) : SV_Target {  
    fixed3 worldNormal = normalize(i.worldNormal);  
    fixed3 worldLightDir =  
normalize(UnityWorldSpaceLightDir(i.worldPos));  
    fixed3 worldViewDir =  
normalize(UnityWorldSpaceViewDir(i.worldPos));  
    fixed3 worldHalfDir = normalize(worldLightDir + worldViewDir);  
  
    fixed4 c = tex2D (_MainTex, i.uv);  
    fixed3 albedo = c.rgb * _Color.rgb;  
  
    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;  
  
    UNITY_LIGHT_ATTENUATION(atten, i, i.worldPos);  
  
    fixed diff = dot(worldNormal, worldLightDir);  
    diff = (diff * 0.5 + 0.5) * atten;  
  
    fixed3 diffuse = _LightColor0.rgb * albedo * tex2D(_Ramp,  
float2(diff, diff)).rgb;  
  
    fixed spec = dot(worldNormal, worldHalfDir);  
    fixed w = fwidth(spec) * 2.0;  
    fixed3 specular = _Specular.rgb * lerp(0, 1, smoothstep(-w, w,  
spec + _SpecularScale  
- 1)) * step(0.0001, _SpecularScale);  
  
    return fixed4(ambient + diffuse + specular, 1.0);  
}
```

首先，我们计算了光照模型中需要的各个方向矢量，并对它们进行了归一化处理。然后，我们计算了材质的反射率`albedo`和环境光照`ambient`。接着，我们使用内置的`UNITYLIGHT_ATTENUATION`宏来计算当前世界坐标下的阴影值。随后，我们计算了半兰伯特漫反射系数，并和阴影值相乘得到最终的漫反射系数。我们使用这个漫反射系数对渐变纹理`_Ramp`进行采样，并将结果和材质的反射率、光照颜色相乘，作为最后的漫反射光照。高光反射的计算和14.1.2节中介绍的方法一致，我们使用`fwidth`对高光区域的边界进行抗锯齿处理，并将计算而得的高光反射系数和高光反射颜色相乘，得到高光反射的光照部分。值得注意的是，我们在最后还使用了`step(0.0001, _SpecularScale)`，这是为了在`_SpecularScale`为0时，可以完全消除高光反射的光照。最后，返回环境光照、漫反射光照和高光反射光照叠加的结果。

(7) 最后，我们为Shader设置了合适的Fallback:

```
Fallback "Diffuse"
```

这对产生正确的阴影投射效果很重要（详见9.4节）。

本节实现的卡通渲染光照模型是一种非常简单的实现。在商业项目中，我们往往需要设计和实现更复杂的光照模型，以得到出色的卡通效果。一个很好的例子是游戏《军团要塞2》（英文名：**Team Fortress 2**）的渲染效果。**Valve**公司在2007年发表了一篇著名的文章<sup>[3]</sup>，解释了他们在实现该游戏时使用的相关技术。

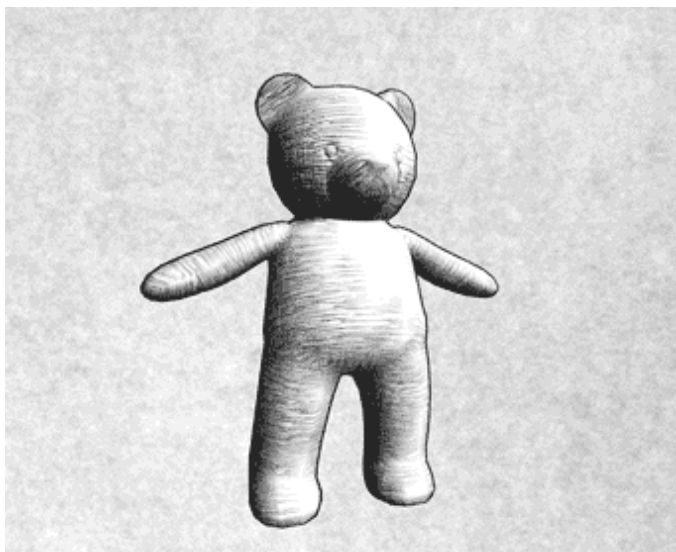
## 14.2 素描风格的渲染

另一个非常流行的非真实感渲染是素描风格的渲染。微软研究院的Praun等人在2001年的SIGGRAPH上发表了一篇非常著名的论文<sup>[4]</sup>。在这篇文章中，他们使用了提前生成的素描纹理来实现实时的素描风格渲染，这些纹理组成了一个**色调艺术映射（Tonal Art Map, TAM）**，如图14.4所示。在图14.4中，从左到右纹理中的笔触逐渐增多，用于模拟不同光照下的漫反射效果，从上到下则对应了每张纹理的多级渐远纹理（mipmaps）。这些多级渐远纹理的生成并不是简单的对上一层纹理进行降采样，而是需要保持笔触之间的间隔，以便更真实地模拟素描效果。



▲ 图14.4 一个TAM的例子（来源：Praun E, et al. Real-time hatching<sup>[4]</sup>）

本节将会实现简化版的论文中提出的算法，我们不考虑多级渐远纹理的生成，而直接使用6张素描纹理进行渲染。在渲染阶段，我们首先在顶点着色阶段计算逐顶点的光照，根据光照结果来决定6张纹理的混合权重，并传递给片元着色器。然后，在片元着色器中根据这些权重来混合6张纹理的采样结果。在学习完本节后，我们会得到类似图14.5的效果。



▲图14.5 素描风格的渲染效果

为此，我们需要进行如下准备工作。

(1) 在Unity中新建一个场景。在本书资源中，该场景名为Scene\_14\_2。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window -> Lighting -> Skybox中去掉场景中的天空盒子。

(2) 新建一个材质。在本书资源中，该材质名为HatchingMat。

(3) 新建一个Unity Shader。在本书资源中，该Unity Shader名为Chapter14-Hatching。把新的Unity Shader赋给第2步中创建的材质。

(4) 在场景中拖曳一个TeddyBear模型，并把第2步中的材质赋给该模型。为了得到更好的效果，我们还把一张纸张图像拖曳到场景中作为背景。

(5) 保存场景。

打开Chapter14-Hatching，进行如下关键修改。

(1) 首先，声明渲染所需的各个属性：

```
Properties {  
    _Color ("Color Tint", Color) = (1, 1, 1, 1)  
    _TileFactor ("Tile Factor", Float) = 1  
    _Outline ("Outline", Range(0, 1)) = 0.1  
    _Hatch0 ("Hatch 0", 2D) = "white" {}  
    _Hatch1 ("Hatch 1", 2D) = "white" {}  
    _Hatch2 ("Hatch 2", 2D) = "white" {}  
    _Hatch3 ("Hatch 3", 2D) = "white" {}  
    _Hatch4 ("Hatch 4", 2D) = "white" {}  
    _Hatch5 ("Hatch 5", 2D) = "white" {}  
}
```

其中，\_Color是用于控制模型颜色的属性。\_TileFactor是纹理的平铺系数，\_TileFactor越大，模型上的素描线条越密，在实现图14.5的过程中，我们把\_TileFactor设置为8。\_Hatch0至\_Hatch5对应了渲染时使用的6张素描纹理，它们的线条密度依次增大。

(2) 由于素描风格往往也需要在物体周围渲染轮廓线，因此我们直接使用14.1节中渲染轮廓线的Pass：

```
SubShader {  
    Tags { "RenderType"="Opaque" "Queue"="Geometry"}  
  
    UsePass "Unity Shaders Book/Chapter 14/Toon Shading/OUTLINE"
```

我们使用UsePass命令调用了14.1节中实现的轮廓线渲染的Pass，Unity Shaders Book/Chapter 14/Toon Shading对应了14.1节中Chapter14-ToonShading文件里Shader的名字，而Unity内部会把Pass的名称全部转成大写格式，所以我们需要在UsePass中使用大写格式的Pass名称。



(3) 下面，我们需要定义光照模型所在的Pass。为了能够正确获取各个光照变量，我们设置了Pass的标签和相关的编译指令：

```
Pass {  
    Tags { "LightMode"="ForwardBase" }  
  
    CGPROGRAM  
  
    #pragma vertex vert  
    #pragma fragment frag  
  
    #pragma multi_compile_fwdbase
```

(4) 由于我们需要在顶点着色器中计算6张纹理的混合权重，我们首先需要在v2f结构体中添加相应的变量：

```
struct v2f {  
    float4 pos : SV_POSITION;  
    float2 uv : TEXCOORD0;  
    fixed3 hatchWeights0 : TEXCOORD1;  
    fixed3 hatchWeights1 : TEXCOORD2;  
    float3 worldPos : TEXCOORD3;  
    SHADOW_COORDS(4)  
};
```

由于一共声明了6张纹理，这意味着需要6个混合权重，我们把它存储在两个fixed3类型的变量（hatchWeights0和hatchWeights1）中。为了添加阴影效果，我们还声明了worldPos变量，并使用SHADOW\_COORDS宏声明了阴影纹理的采样坐标。

(5) 然后，我们定义了关键的顶点着色器：

```
v2f vert(a2v v) {  
    v2f o;  
  
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);  
  
    o.uv = v.texcoord.xy * _TileFactor;
```

```

fixed3 worldLightDir = normalize(WorldSpaceLightDir(v.vertex));
fixed3 worldNormal = UnityObjectToWorldNormal(v.normal);
fixed diff = max(0, dot(worldLightDir, worldNormal));

o.hatchWeights0 = fixed3(0, 0, 0);
o.hatchWeights1 = fixed3(0, 0, 0);

float hatchFactor = diff * 7.0;

if (hatchFactor > 6.0) {
    // Pure white, do nothing
} else if (hatchFactor > 5.0) {
    o.hatchWeights0.x = hatchFactor - 5.0;
} else if (hatchFactor > 4.0) {
    o.hatchWeights0.x = hatchFactor - 4.0;
    o.hatchWeights0.y = 1.0 - o.hatchWeights0.x;
} else if (hatchFactor > 3.0) {
    o.hatchWeights0.y = hatchFactor - 3.0;
    o.hatchWeights0.z = 1.0 - o.hatchWeights0.y;
} else if (hatchFactor > 2.0) {
    o.hatchWeights0.z = hatchFactor - 2.0;
    o.hatchWeights1.x = 1.0 - o.hatchWeights0.z;
} else if (hatchFactor > 1.0) {
    o.hatchWeights1.x = hatchFactor - 1.0;
    o.hatchWeights1.y = 1.0 - o.hatchWeights1.x;
} else {
    o.hatchWeights1.y = hatchFactor;
    o.hatchWeights1.z = 1.0 - o.hatchWeights1.y;
}

o.worldPos = mul(_Object2World, v.vertex).xyz;

TRANSFER_SHADOW(o);

return o;
}

```

我们首先对顶点进行了基本的坐标变换。然后，使用`_TileFactor`得到了纹理采样坐标。在计算6张纹理的混合权重之前，我们首先需要计算逐顶点光照。因此，我们使用世界空间下的光照方向和法线方向得到漫反射系数`diff`。之后，我们把权重值初始化为0，并把`diff`缩放到[0, 7]范围，得到`hatchFactor`。我们把[0, 7]的区间均匀划分为7个子区间，通过判断`hatchFactor`所处的子区间来计算对应的纹理混合权重。最后，

我们计算了顶点的世界坐标，并使用TRANSFER\_SHADOW宏来计算阴影纹理的采样坐标。

(6) 接下来，定义片元着色器部分：

```
fixed4 frag(v2f i) : SV_Target {
    fixed4 hatchTex0 = tex2D(_Hatch0, i.uv) * i.hatchWeights0.x;
    fixed4 hatchTex1 = tex2D(_Hatch1, i.uv) * i.hatchWeights0.y;
    fixed4 hatchTex2 = tex2D(_Hatch2, i.uv) * i.hatchWeights0.z;
    fixed4 hatchTex3 = tex2D(_Hatch3, i.uv) * i.hatchWeights1.x;
    fixed4 hatchTex4 = tex2D(_Hatch4, i.uv) * i.hatchWeights1.y;
    fixed4 hatchTex5 = tex2D(_Hatch5, i.uv) * i.hatchWeights1.z;
    fixed4 whiteColor = fixed4(1, 1, 1, 1) * (1 - i.hatchWeights0.x
- i.hatchWeights0.y - i.hatchWeights0.z -
                                i.hatchWeights1.x - i.hatchWeights1.y -
i.hatchWeights1.z);

    fixed4 hatchColor = hatchTex0 + hatchTex1 + hatchTex2 +
hatchTex3 + hatchTex4 + hatchTex5 + whiteColor;

    UNITY_LIGHT_ATTENUATION(atten, i, i.worldPos);

    return fixed4(hatchColor.rgb * _Color.rgb * atten, 1.0);
}
```

当得到了6张纹理的混合权重后，我们对每张纹理进行采样并和它们对应的权重值相乘得到每张纹理的采样颜色。我们还计算了纯白在渲染中的贡献度，这是通过从1中减去所有6张纹理的权重来得到的。这是因为素描中往往有留白的部分，因此我们希望在最后的渲染中光照最亮的部分是纯白色的。最后，我们混合了各个颜色值，并和阴影值atten、模型颜色\_Color相乘后返回最终的渲染结果。

(7) 最后，我们设置了合适的Fallback：

```
Fallback "Diffuse"
```

读者也可以生成与本例不同的素描纹理，具体方法可以参见论文<sup>[4]</sup>。这篇博文 (<https://alastaira.wordpress.com/2013/11/01/hand-drawn-shaders-and-creating-tonal-art-maps/>) 中还介绍了一种使用Photoshop等软件创建相似的素描纹理的方法。

## 14.3 扩展阅读

在工业界，非真实感渲染已被应用到很多成功的游戏中，除了之前提及的《大神》和《军团要塞2》外，还有最近的《海岛奇兵》《三国志》等游戏都可以看到非真实感渲染的身影。在学术界，有更多出色的非真实感渲染的工作被提了出来。读者可以在国际讨论会NPAR (Non-Photorealistic Animation and Rendering) 上找到许多关于非真实感渲染的论文。浙江大学的耿卫东教授编纂的书籍《艺术化绘制的图形学原理与方法》(英文名: *The Algorithms and Principles of Non-photorealistic Graphics*)<sup>[5]</sup>，也是非常好的学习材料。这本书概述了近年来非真实感渲染在各个领域的发展，并简述了许多有重要贡献的算法过程，是一本非常好的参考书籍。

在Unity的资源商店中，也有许多优秀的非真实感渲染资源。例如，**Toon Shader Free**

(<https://www.assetstore.unity3d.com/cn/#!/content/21288>) 是一个免费的卡通资源包，里面实现了包括轮廓线渲染等卡通风格的渲染。**Toon Styles Shader Pack** (<https://www.assetstore.unity3d.com/cn/#!/content/7212>) 是一个需要收费的卡通资源包，它包含了更多的卡通风格的Unity Shader。**Hand-Drawn Shader Pack**

(<https://www.assetstore.unity3d.com/cn/#!/content/12465>) 同样是一个需

要收费的非真实感渲染效果包，它包含了诸如铅笔渲染、蜡笔渲染等多种手绘风格的非真实感渲染效果。

## 14.4 参考文献

[1] Gooch A, Gooch B, Shirley P, et al. A non-photorealistic lighting model for automatic technical illustration[C]//Proceedings of the 25th annual conference on Computer graphics and interactive techniques. ACM, 1998: 447-452。

[2] Anjyo K, Hiramitsu K. Stylized highlights for cartoon rendering and animation[J]. Computer Graphics and Applications, IEEE, 2003, 23(4): 54-61。

[3] Mitchell J, Francke M, Eng D. Illustrative rendering in Team Fortress 2[C]//Proceedings of the 5th international symposium on Non-photorealistic animation and rendering. ACM, 2007: 71-76。

[4] Praun E, Hoppe H, Webb M, et al. Real-time hatching[C]//Proceedings of the 28th annual conference on Computer graphics and interactive techniques. ACM, 2001: 581。

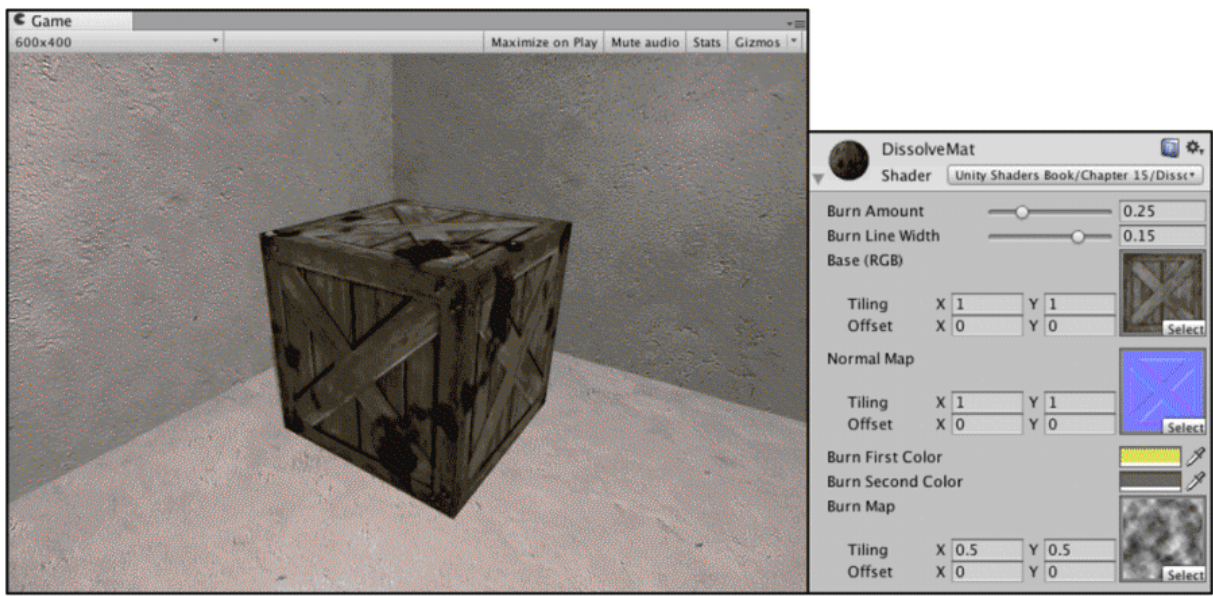
[5] Geng W. The Algorithms and Principles of Non-photorealistic Graphics: Artistic Rendering and Cartoon Animation[M]. Springer Science & Business Media, 2011。

## 第15章 使用噪声

很多时候，向规则的事物里添加一些“杂乱无章”的效果往往会有意想不到的效果。而这些“杂乱无章”的效果来源就是噪声。在本章中，我们将会学习如何使用噪声来模拟各种看似“神奇”的特效。在15.1节中，我们将使用一张噪声纹理来模拟火焰的消融效果。15.2节则把噪声应用在模拟水面的波动上，从而产生波光粼粼的视觉效果。在15.3节中，我们会回顾13.3节中实现的全局雾效，并向其中添加噪声来模拟不均匀的飘渺雾效。

### 15.1 消融效果

**消融（dissolve）**效果常见于游戏中的角色死亡、地图烧毁等效果。在这些效果中，消融往往从不同的区域开始，并向看似随机的方向扩张，最后整个物体都将消失不见。在本节中，我们将学习如何在Unity中实现这种效果。在学习完本节后，我们可以得到类似图15.1中的效果。



▲图15.1 箱子的消融效果

要实现图15.1中的效果，原理非常简单，概括来说就是噪声纹理+透明度测试。我们使用对噪声纹理采样的结果和某个控制消融程度的阈值比较，如果小于阈值，就使用clip函数把它对应的像素裁剪掉，这些部分就对应了图中被“烧毁”的区域。而镂空区域边缘的烧焦效果则是将两种颜色混合，再用pow函数处理后，与原纹理颜色混合后的结果。

为了实现上述消融效果，我们首先进行如下准备工作。

(1) 在Unity中新建一个场景。在本书资源中，该场景名为Scene\_15\_1。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window -> Lighting -> Skybox中去掉场景中的天空盒子。

(2) 新建一个材质。在本书资源中，该材质名为DissolveMat。



(3) 新建一个Unity Shader。在本书资源中，该Unity Shader名为Chapter15-Dissolve。把新的Unity Shader赋给第2步中创建的材质。

(4) 我们需要搭建一个测试消融的场景。在本书资源的实现中，我们构建了一个包含3面墙的房间，并放置了一个立方体。把第2步创建的材质拖曳给立方体。

(5) 保存场景。

打开Chapter15-Dissolve，删除原有代码，进行如下关键修改。

(1) 首先，声明消融效果需要的各个属性：

```
Properties {  
    _BurnAmount ("Burn Amount", Range(0.0, 1.0)) = 0.0  
    _LineWidth ("Burn Line Width", Range(0.0, 0.2)) = 0.1  
    _MainTex ("Base (RGB)", 2D) = "white" {}  
    _BumpMap ("Normal Map", 2D) = "bump" {}  
    _BurnFirstColor ("Burn First Color", Color) = (1, 0, 0, 1)  
    _BurnSecondColor ("Burn Second Color", Color) = (1, 0, 0, 1)  
    _BurnMap ("Burn Map", 2D) = "white" {}  
}
```

\_BurnAmount属性用于控制消融程度，当值为0时，物体为正常效果，当值为1时，物体会完全消融。\_LineWidth属性用于控制模拟烧焦效果时的线宽，它的值越大，火焰边缘的蔓延范围越广。\_MainTex和\_BumpMap分别对应了物体原本的漫反射纹理和法线纹理。

\_BurnFirstColor和\_BurnSecondColor对应了火焰边缘的两种颜色值。

\_BurnMap则是关键的噪声纹理。

(2) 我们在SubShader块中定义消融所需的Pass：

```

Pass {
    Tags { "LightMode"="ForwardBase" }

    Cull Off

    CGPROGRAM

    #include "Lighting.cginc"
    #include "AutoLight.cginc"

    #pragma multi_compile_fwdbase

```

为了得到正确的光照，我们设置了Pass的LightMode和multi\_compile\_fwdbase的编译指令。值得注意的是，我们还使用Cull命令关闭了该Shader的面片剔除，也就是说，模型的正面和背面都会被渲染。这是因为，消融会导致裸露模型内部的构造，如果只渲染正面会出现错误的结果。

### (3) 定义顶点着色器:

```

struct v2f {
    float4 pos : SV_POSITION;
    float2 uvMainTex : TEXCOORD0;
    float2 uvBumpMap : TEXCOORD1;
    float2 uvBurnMap : TEXCOORD2;
    float3 lightDir : TEXCOORD3;
    float3 worldPos : TEXCOORD4;
    SHADOW_COORDS(5)
};

v2f vert(a2v v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.uvMainTex = TRANSFORM_TEX(v.texcoord, _MainTex);
    o.uvBumpMap = TRANSFORM_TEX(v.texcoord, _BumpMap);
    o.uvBurnMap = TRANSFORM_TEX(v.texcoord, _BurnMap);

    TANGENT_SPACE_ROTATION;
    o.lightDir = mul(rotation, ObjSpaceLightDir(v.vertex)).xyz;

    o.worldPos = mul(_Object2World, v.vertex).xyz;

```

```
TRANSFER_SHADOW(o);  
  
    return o;  
}
```

顶点着色器的代码很常规。我们使用宏**TRANSFORM\_TEX**计算了三张纹理对应的纹理坐标，再把光源方向从模型空间变换到了切线空间。最后，为了得到阴影信息，计算了世界空间下的顶点位置和阴影纹理的采样坐标（使用了**TRANSFER\_SHADOW**宏）。具体原理可参见9.4节。

(4) 我们还需要实现片元着色器来模拟消融效果:

```
fixed4 frag(v2f i) : SV_Target {  
    fixed3 burn = tex2D(_BurnMap, i.uvBurnMap).rgb;  
  
    clip(burn.r - _BurnAmount);  
  
    float3 tangentLightDir = normalize(i.lightDir);  
    fixed3 tangentNormal = UnpackNormal(tex2D(_BumpMap,  
i.uvBumpMap));  
  
    fixed3 albedo = tex2D(_MainTex, i.uvMainTex).rgb;  
  
    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;  
  
    fixed3 diffuse = _LightColor0.rgb * albedo * max(0,  
dot(tangentNormal, tangentLightDir));  
  
    fixed t = 1 - smoothstep(0.0, _LineWidth, burn.r -  
_BurnAmount);  
    fixed3 burnColor = lerp(_BurnFirstColor, _BurnSecondColor, t);  
    burnColor = pow(burnColor, 5);  
  
    UNITY_LIGHT_ATTENUATION(atten, i, i.worldPos);  
    fixed3 finalColor = lerp(ambient + diffuse * atten, burnColor,  
t * step(0.0001, _BurnAmount));  
  
    return fixed4(finalColor, 1);  
}
```

我们首先对噪声纹理进行采样，并将采样结果和用于控制消融程度的属性\_BurnAmount相减，传递给clip函数。当结果小于0时，该像素将会被剔除，从而不会显示到屏幕上。如果通过了测试，则进行正常的光照计算。我们首先根据漫反射纹理得到材质的反射率albedo，并由此计算得到环境光照，进而得到漫反射光照。然后，我们计算了烧焦颜色burnColor。我们想要在宽度为\_LineWidth的范围内模拟一个烧焦的颜色变化，第一步就使用了smoothstep函数来计算混合系数 $t$ 。当 $t$ 值为1时，表明该像素位于消融的边界处，当 $t$ 值为0时，表明该像素为正常的模型颜色，而中间的插值则表示需要模拟一个烧焦效果。我们首先用 $t$ 来混合两种火焰颜色\_BurnFirstColor和\_BurnSecondColor，为了让效果更接近烧焦的痕迹，我们还使用pow函数对结果进行处理。然后，我们再次使用 $t$ 来混合正常的光照颜色（环境光+漫反射）和烧焦颜色。我们这里又使用了step函数来保证当\_BurnAmount为0时，不显示任何消融效果。最后，返回混合后的颜色值finalColor。

(5) 与之前的实现不同，我们在本例中还定义了一个用于投射阴影的Pass。正如我们在9.4.5节中的解释一样，使用透明度测试的物体的阴影需要特别处理，如果仍然使用普通的阴影Pass，那么被剔除的区域仍然会向其他物体投射阴影，造成“穿帮”。为了让物体的阴影也能配合透明度测试产生正确的效果，我们需要自定义一个投射阴影的Pass：

```
// Pass to render object as a shadow caster
Pass {
    Tags { "LightMode" = "ShadowCaster" }

    CGPROGRAM

    #pragma vertex vert
    #pragma fragment frag

    #pragma multi_compile_shadowcaster
```

在Unity中，用于投射阴影的Pass的LightMode需要被设置为ShadowCaster，同时，还需要使用#pragma multi\_compile\_shadowcaster指明它需要的编译指令。

顶点着色器和片元着色器的代码很简单：

```
struct v2f {
    V2F_SHADOW_CASTER;
    float2 uvBurnMap : TEXCOORD1;
};

v2f vert(appdata_base v) {
    v2f o;

    TRANSFER_SHADOW_CASTER_NORMALOFFSET(o)

    o.uvBurnMap = TRANSFORM_TEX(v.texcoord, _BurnMap);

    return o;
}

fixed4 frag(v2f i) : SV_Target {
    fixed3 burn = tex2D(_BurnMap, i.uvBurnMap).rgb;

    clip(burn.r - _BurnAmount);

    SHADOW_CASTER_FRAGMENT(i)
}
```

阴影投射的重点在于我们需要按正常Pass的处理来剔除片元或进行顶点动画，以便阴影可以和物体正常渲染的结果相匹配。在自定义的阴影投射的Pass中，我们通常会使用Unity提供的内置宏

**V2FSHADOW\_CASTER**、

**TRANSFER\_SHADOW\_CASTER\_NORMALOFFSET**（旧版本中会使用**TRANSFER\_SHADOW\_CASTER**）和**SHADOW\_CASTER\_FRAGMENT**来帮助我们计算阴影投射时需要的各种变量，而我们可以只关注自定义计算的部分。在上面的代码中，我们首先在v2f结构体中利用

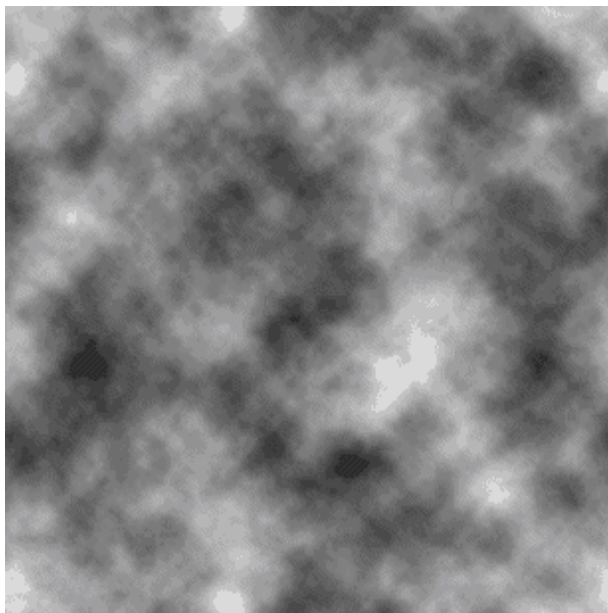
`V2F_SHADOW_CASTER`来定义阴影投射需要定义的变量。随后，在顶点着色器中，我们使用

`TRANSFER_SHADOW_CASTER_NORMALOFFSET`来填充

`V2F_SHADOW_CASTER`在背后声明的一些变量，这是由Unity在背后为我们完成的。我们需要在顶点着色器中关注自定义的计算部分，这里指的就是我们需要计算噪声纹理的采样坐标`uvBurnMap`。在片元着色器中，我们首先按之前的处理方法使用噪声纹理的采样结果来剔除片元，最后再利用`SHADOW_CASTER_FRAGMENT`来让Unity为我们完成阴影投射的部分，把结果输出到深度图和阴影映射纹理中。

通过Unity提供的这3个内置宏（在`UnityCG.cginc`文件中被定义），我们可以方便地自定义需要的阴影投射的Pass，但由于这些宏需要使用一些特定的输入变量，因此我们需要保证为它们提供了这些变量。例如，`TRANSFER_SHADOW_CASTER_NORMALOFFSET`会使用名称`v`作为输入结构体，`v`中需要包含顶点位置`v.vertex`和顶点法线`v.normal`的信息，我们可以直接使用内置的`appdata_base`结构体，它包含了这些必需的顶点变量。如果我们需要进行顶点动画，可以在顶点着色器中直接修改`v.vertex`，再传递给`TRANSFER_SHADOW_CASTER_NORMALOFFSET`即可（可参见11.3.3节）。

在本例中，我们使用的噪声纹理（对应本书资源的`Assets/Textures/Chapter15/Burn_Noise.png`）如图15.2所示。把它拖曳到材质的`_BurnMap`属性上，再调整材质的`_BurnAmount`属性，就可以看到木箱逐渐消融的效果。在本书资源的实现中，我们实现了一个辅助脚本，用来随时间调整材质的`_BurnAmount`值，因此，当读者单击运行后，也可以看到消融的动画效果。



▲图15.2 消融效果使用的噪声纹理

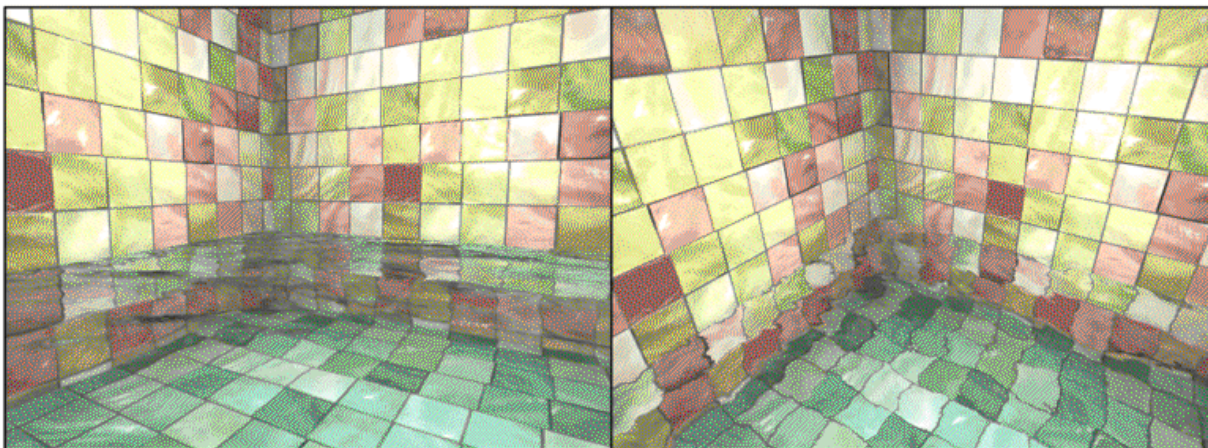
使用不同的噪声和纹理属性（即材质面板上纹理的Tiling和Offset值）都会得到不同的消融效果。因此，要想得到好的消融效果，也需要美术人员提供合适的噪声纹理来配合。

## 15.2 水波效果

在模拟实时水面的过程中，我们往往也会使用噪声纹理。此时，噪声纹理通常会用作一个高度图，以不断修改水面的法线方向。为了模拟水不断流动的效果，我们会使用和时间相关的变量来对噪声纹理进行采样，当得到法线信息后，再进行正常的反射+折射计算，得到最后的水面波动效果。

在本节中，我们将会使用一个由噪声纹理得到的法线贴图，实现一个包含菲涅耳反射（详见10.1.5节）的水面效果，如图15.3所示。





▲ 图15.3 包含菲涅耳反射的水面波动效果。在左边中，视角方向和水面法线的夹角越大，反射效果越强。在右边中，视角方向和水面法线的夹角越大，折射效果越强

我们曾在10.2.2节介绍过如何使用反射和折射来模拟一个透明玻璃的效果。本节使用的Shader和10.2.2节中的实现基本相同。我们使用一张立方体纹理（Cubemap）作为环境纹理，模拟反射。为了模拟折射效果，我们使用GrabPass来获取当前屏幕的渲染纹理，并使用切线空间下的法线方向对像素的屏幕坐标进行偏移，再使用该坐标对渲染纹理进行屏幕采样，从而模拟近似的折射效果。与10.2.2节中的实现不同的是，水波的法线纹理是由一张噪声纹理生成而得，而且会随着时间变化不断平移，模拟波光粼粼的效果。除此之外，我们没有使用一个定值来混合反射和折射颜色，而是使用之前提到的菲涅耳系数来动态决定混合系数。我们使用如下公式来计算菲涅耳系数：

$$fresnel = \text{pow}(1 - \max(0, \mathbf{v} \cdot \mathbf{n}), 4)$$

其中， $\mathbf{v}$ 和 $\mathbf{n}$ 分别对应了视角方向和法线方向。它们之间的夹角越小， $fresnel$ 值越小，反射越弱，折射越强。菲涅耳系数还经常会用于边缘光照的计算中。

为此，我们需要做如下准备工作。

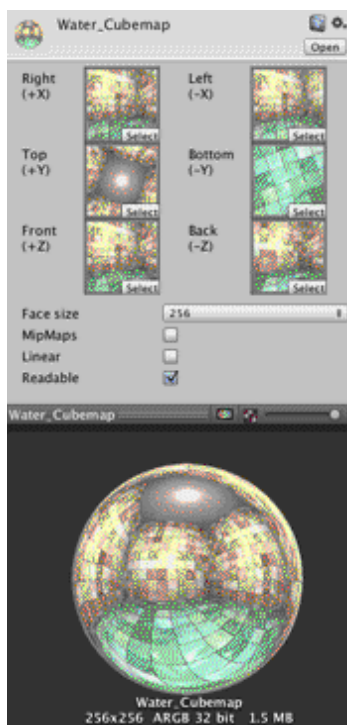
(1) 新建一个场景。在本书资源中，该场景名为**Scene\_15\_2**。在**Unity 5.2**中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在**Window -> Lighting -> Skybox**中去掉场景中的天空盒子。

(2) 新建一个材质。在本书资源中，该材质名为**WaterWaveMat**。

(3) 新建一个**Unity Shader**。在本书资源中，该**Shader**名为**Chapter15-WaterWave**。把新的**Shader**赋给第2步中创建的材质。

(4) 构建一个测试水波效果的场景。在本书资源的实现中，我们构建了一个由6面墙围成的封闭房间，它们都使用了我们在9.5节中创建的标准材质。我们还在房间中放置了一个平面来模拟水面。把第2步中创建的材质赋给该平面。

(5) 为了得到本场景适用的环境纹理，我们使用了10.1.2节中实现的创建立方体纹理的脚本（通过**GameObject -> Render into Cubemap**打开编辑窗口）来创建它，如图15.4所示。在本书资源中，该**Cubemap**名为**Water\_Cubemap**。



▲图15.4 本例使用的立方体纹理

完成准备工作后，打开Chapter15-WaterWave，对它进行如下关键修改。

(1) 首先，我们需要声明该Shader使用的各个属性：

```
Properties {
    _Color ("Main Color", Color) = (0, 0.15, 0.115, 1)
    _MainTex ("Base (RGB)", 2D) = "white" {}
    _WaveMap ("Wave Map", 2D) = "bump" {}
    _Cubemap ("Environment Cubemap", Cube) = "_Skybox" {}
    _WaveXSpeed ("Wave Horizontal Speed", Range(-0.1, 0.1)) = 0.01
    _WaveYSpeed ("Wave Vertical Speed", Range(-0.1, 0.1)) = 0.01
    _Distortion ("Distortion", Range(0, 100)) = 10
}
```

其中，`_Color`用于控制水面颜色；`_MainTex`是水面波纹材质纹理，默认为白色纹理；`_WaveMap`是一个由噪声纹理生成的法线纹理；`_Cubemap`是用于模拟反射的立方体纹理；`_Distortion`则用于控制模拟

折射时图像的扭曲程度；\_WaveXSpeed和\_WaveYSpeed分别用于控制法线纹理在X和Y方向上的平移速度。

(2) 定义相应的渲染队列，并使用GrabPass来获取屏幕图像：

```
SubShader {  
    // We must be transparent, so other objects are drawn before  
    this one.  
    Tags { "Queue"="Transparent" "RenderType"="Opaque" }  
  
    // This pass grabs the screen behind the object into a texture.  
    // We can access the result in the next pass as _RefractionTex  
    GrabPass { "_RefractionTex" }
```

我们首先在SubShader的标签中将渲染队列设置成Transparent，并把后面的RenderType设置为Opaque。把Queue设置成Transparent可以确保该物体渲染时，其他所有不透明物体都已经被渲染到屏幕上了，否则就可能无法正确得到“透过水面看到的图像”。而设置RenderType则是为了在使用着色器替换（Shader Replacement）时，该物体可以在需要时被正确渲染。这通常发生在我们需要得到摄像机的深度和法线纹理时，这在第13章中介绍过。随后，我们通过关键词GrabPass定义了一个抓取屏幕图像的Pass。在这个Pass中我们定义了一个字符串，该字符串内部的名称决定了抓取得到的屏幕图像将会被存入哪个纹理中（可参见10.2.2节）。

(3) 定义渲染水面所需的Pass。为了在Shader中访问各个属性，我们首先需要定义它们对应的变量：

```
fixed4 _Color;  
sampler2D _MainTex;  
float4 _MainTex_ST;  
sampler2D _WaveMap;  
float4 _WaveMap_ST;  
samplerCUBE _Cubemap;
```

```
fixed _WaveXSpeed;  
fixed _WaveYSpeed;  
float _Distortion;  
sampler2D _RefractionTex;  
float4 _RefractionTex_TexelSize;
```

需要注意的是，我们还定义了`_RefractionTex`和`_RefractionTex_TexelSize`变量，这对应了在使用`GrabPass`时，指定的纹理名称。`_RefractionTex_TexelSize`可以让我们得到该纹理的纹素大小，例如一个大小为 $256 \times 512$ 的纹理，它的纹素大小为 $(1/256, 1/512)$ 。我们需要在对屏幕图像的采样坐标进行偏移时使用该变量。

(4) 定义顶点着色器，这和10.2.2节中的实现完全一样：

```
struct v2f {  
    float4 pos : SV_POSITION;  
    float4 scrPos : TEXCOORD0;  
    float4 uv : TEXCOORD1;  
    float4 TtoW0 : TEXCOORD2;  
    float4 TtoW1 : TEXCOORD3;  
    float4 TtoW2 : TEXCOORD4;  
};  
  
v2f vert(a2v v) {  
    v2f o;  
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);  
  
    o.scrPos = ComputeGrabScreenPos(o.pos);  
  
    o.uv.xy = TRANSFORM_TEX(v.texcoord, _MainTex);  
    o.uv.zw = TRANSFORM_TEX(v.texcoord, _WaveMap);  
  
    float3 worldPos = mul(_Object2World, v.vertex).xyz;  
    fixed3 worldNormal = UnityObjectToWorldNormal(v.normal);  
    fixed3 worldTangent = UnityObjectToWorldDir(v.tangent.xyz);  
    fixed3 worldBinormal = cross(worldNormal, worldTangent) *  
v.tangent.w;  
  
    o.TtoW0 = float4(worldTangent.x, worldBinormal.x,  
worldNormal.x, worldPos.x);  
    o.TtoW1 = float4(worldTangent.y, worldBinormal.y,  
worldNormal.y, worldPos.y);  
    o.TtoW2 = float4(worldTangent.z, worldBinormal.z,  
worldNormal.z, worldPos.z);
```

```
    return o;  
}
```

在进行了必要的顶点坐标变换后，我们通过调用 `ComputeGrabScreenPos` 来得到对应被抓取屏幕图像的采样坐标。读者可以在 `UnityCG.cginc` 文件中找到它的声明，它的主要代码和 `ComputeScreenPos` 基本类似，最大的不同是针对平台差异造成的采样坐标问题（见5.6.1节）进行了处理。接着，我们计算了 `_MainTex` 和 `_BumpMap` 的采样坐标，并把它们分别存储在一个 `float4` 类型变量的 `xy` 和 `zw` 分量中。由于我们需要在片元着色器中把法线方向从切线空间（由法线纹理采样得到）变换到世界空间下，以便对 `Cubemap` 进行采样，因此，我们需要在这里计算该顶点对应的从切线空间到世界空间的变换矩阵，并把该矩阵的每一行分别存储在 `TtoW0`、`TtoW1` 和 `TtoW2` 的 `xyz` 分量中。这里面使用的数学方法就是，得到切线空间下的3个坐标轴（`x`、`y`、`z` 轴分别对应了切线、副切线和法线的方向）在世界空间下的表示，再把它们依次按列组成一个变换矩阵即可。`TtoW0` 等值的 `w` 分量同样被利用起来，用于存储世界空间下的顶点坐标。

#### (5) 定义片元着色器：

```
fixed4 frag(v2f i) : SV_Target {  
    float3 worldPos = float3(i.TtoW0.w, i.TtoW1.w, i.TtoW2.w);  
    fixed3 viewDir = normalize(UnityWorldSpaceViewDir(worldPos));  
    float2 speed = _Time.y * float2(_WaveXSpeed, _WaveYSpeed);  
  
    // Get the normal in tangent space  
    fixed3 bump1 = UnpackNormal(tex2D(_WaveMap, i.uv.zw +  
speed)).rgb;  
    fixed3 bump2 = UnpackNormal(tex2D(_WaveMap, i.uv.zw -  
speed)).rgb;  
    fixed3 bump = normalize(bump1 + bump2);  
  
    // Compute the offset in tangent space  
    float2 offset = bump.xy * _Distortion *  
}
```

```

_RefractionTex_TexelSize.xy;
    i.scrPos.xy = offset * i.scrPos.z + i.scrPos.xy;
    fixed3 refrCol = tex2D( _RefractionTex,
i.scrPos.xy/i.scrPos.w).rgb;

    // Convert the normal to world space
    bump = normalize(half3(dot(i.TtoW0.xyz, bump), dot(i.TtoW1.xyz,
bump), dot(i.TtoW2.xyz, bump)));
    fixed4 texColor = tex2D(_MainTex, i.uv.xy + speed);
    fixed3 reflDir = reflect(-viewDir, bump);
    fixed3 reflCol = texCUBE(_Cubemap, reflDir).rgb * texColor.rgb
* _Color.rgb;

    fixed fresnel = pow(1 - saturate(dot(viewDir, bump)), 4);
    fixed3 finalColor = reflCol * fresnel + refrCol * (1 -
fresnel);

    return fixed4(finalColor, 1);
}

```

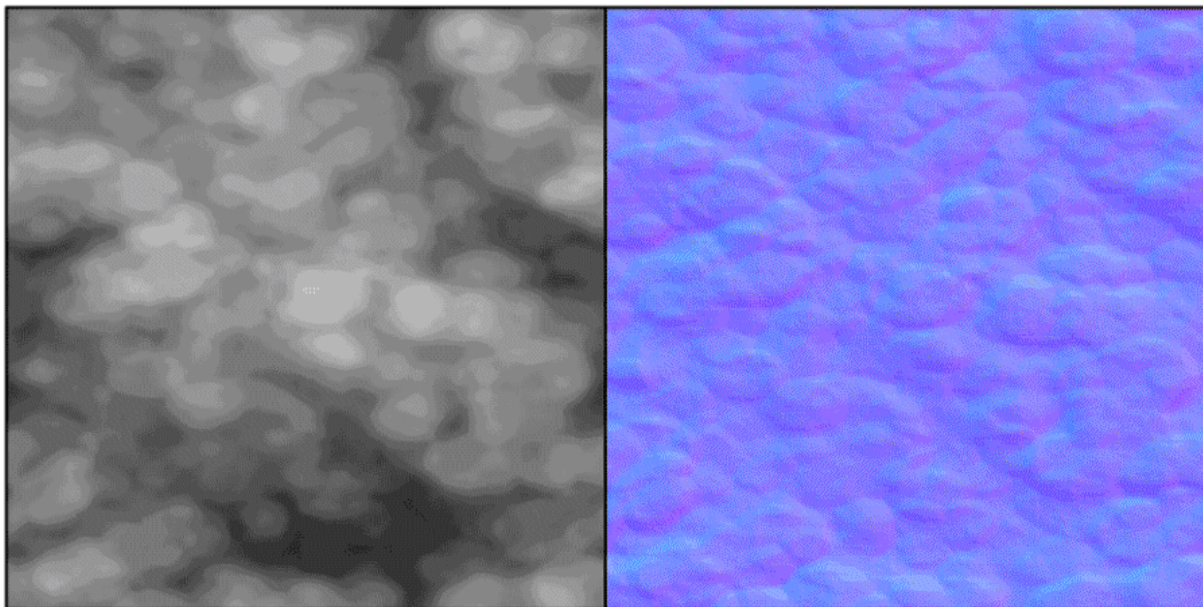
我们首先通过TtoW0等变量的w分量得到世界坐标，并用该值得到该片元对应的视角方向。除此之外，我们还使用内置的\_Time.y变量和\_WaveXSpeed、\_WaveYSpeed属性计算了法线纹理的当前偏移量，并利用该值对法线纹理进行两次采样（这是为了模拟两层交叉的水面波动的效果），对两次结果相加并归一化后得到切线空间下的法线方向。然后，和10.2.2节中的处理一样，我们使用该值和\_Distortion属性以及\_RefractionTex\_TexelSize来对屏幕图像的采样坐标进行偏移，模拟折射效果。\_Distortion值越大，偏移量越大，水面背后的物体看起来变形程度越大。在这里，我们选择使用切线空间下的法线方向来进行偏移，是因为该空间下的法线可以反映顶点局部空间下的法线方向。需要注意的是，在计算偏移后的屏幕坐标时，我们把偏移量和屏幕坐标的z分量相乘，这是为了模拟深度越大、折射程度越大的效果。如果读者不希望产生这样的效果，可以直接把偏移值叠加到屏幕坐标上。随后，我们对scrPos进行了透视除法，再使用该坐标对抓取的屏幕图像\_RefractionTex进行采样，得到模拟的折射颜色。



之后，我们把法线方向从切线空间变换到了世界空间下（使用变换矩阵的每一行，即 $TtoW0$ 、 $TtoW1$ 和 $TtoW2$ ，分别和法线方向点乘，构成新的法线方向），并据此得到视角方向相对于法线方向的反射方向。随后，使用反射方向对Cubemap进行采样，并把结果和主纹理颜色相乘后得到反射颜色。我们也对主纹理进行了纹理动画，以模拟水波的效果。

为了混合折射和反射颜色，我们随后计算了菲涅耳系数。我们使用之前的公式来计算菲涅耳系数，并据此来混合折射和反射颜色，作为最终的输出颜色。

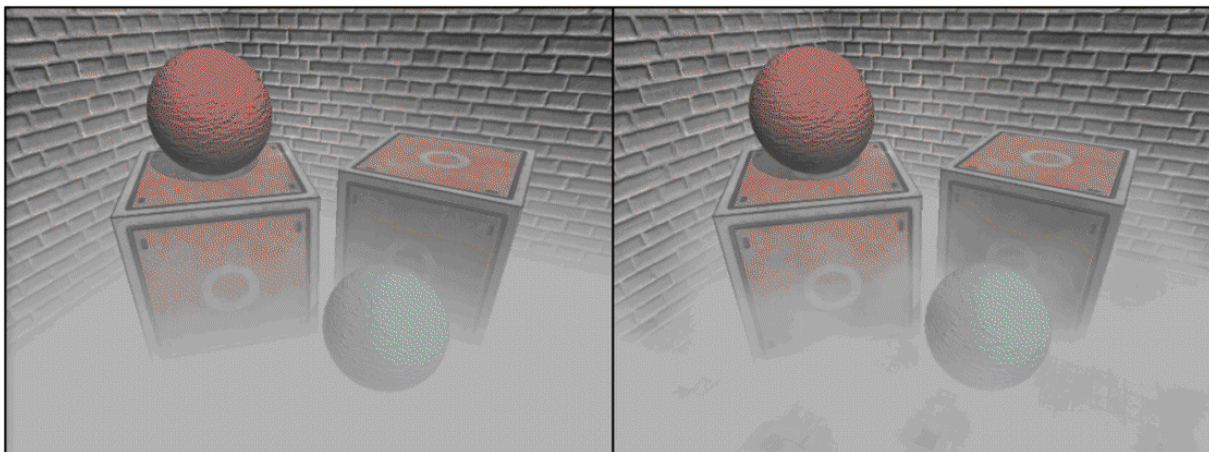
在本例中，我们使用的噪声纹理（对应本书资源的Assets/Textures/Chapter15/Water\_Noise.png）如图15.5左图所示。由于在本例中，我们需要的是一张法线纹理，因此我们可以从该噪声纹理的灰度值中生成需要的法线信息，这是通过在它的纹理面板中把纹理类型设置为**Normal map**，并选中**Create from grayscale**来完成的。最后生成的法线纹理如图15.5右图所示。我们把生成的法线纹理拖曳到材质的\_WaveMap属性上，再单击运行后，就可以看到水面波动的效果了。



▲图15.5 水波效果使用的噪声纹理左边：噪声纹理的灰度图，右边：由左边生成的法线纹理

## 5.3 再谈全局雾效

我们在13.3节讲到了如何使用深度纹理来实现一种基于屏幕后处理的全局雾效。我们由深度纹理重建每个像素在世界空间下的位置，再使用一个基于高度的公式来计算雾效的混合系数，最后使用该系数来混合雾的颜色和原屏幕颜色。13.3节的实现效果是一个基于高度的均匀雾效，即在同一个高度上，雾的浓度是相同的，如图15.6左图所示。然而，一些时候我们希望可以模拟一种不均匀的雾效，同时让雾不断飘动，使雾看起来更加飘渺，如图15.6右图所示。而这就可以通过使用一张噪声纹理来实现。



▲图15.6 左边：均匀雾效，右边：使用噪声纹理后的非均匀雾效

本节的实现非常简单，绝大多数代码和13.3节中的完全一样，我们只是添加了噪声相关的参数和属性，并在Shader的片元着色器中对高度的计算添加了噪声的影响。为了完整性，我们会给出本节使用的脚本和Shader的实现，但其中使用的原理不再赘述，读者可参见13.3节。

我们首先需要进行如下准备工作。

(1) 新建一个场景。在本书资源中，该场景名为Scene\_15\_3。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window -> Lighting -> Skybox中去掉场景中的天空盒子。

(2) 我们需要搭建一个测试雾效的场景。在本书资源的实现中，我们构建了一个包含3面墙的房间，并放置了两个立方体和两个球体，它们都使用了我们在9.5节中创建的标准材质。

(3) 新建一个脚本。在本书资源中，该脚本名为FogWithNoise.cs。把该脚本拖曳到摄像机上。

(4) 新建一个Unity Shader。在本书资源中，该Shader名为Chapter15-FogWithNoise。

我们首先来编写FogWithNoise.cs脚本。打开该脚本，并进行如下修改。

(1) 首先，继承12.1节中创建的基类：

```
public class FogWithNoise : PostEffectsBase {
```

(2) 声明该效果需要的Shader，并据此创建相应的材质：

```
public Shader fogShader;
private Material fogMaterial = null;

public Material material {
    get {
        fogMaterial = CheckShaderAndCreateMaterial(fogShader,
        fogMaterial);
        return fogMaterial;
    }
}
```

(3) 在本节中，我们需要获取摄像机的相关参数，如近裁剪平面的距离、FOV等，同时还需要获取摄像机在世界空间下的前方、上方和右方等方向，因此我们用两个变量存储摄像机的Camera组件和Transform组件：

```
private Camera myCamera;
public Camera camera {
    get {
        if (myCamera == null) {
            myCamera = GetComponent<Camera>();
        }
        return myCamera;
    }
}
```

```
private Transform myCameraTransform;
public Transform cameraTransform {
    get {
        if (myCameraTransform == null) {
            myCameraTransform = camera.transform;
        }

        return myCameraTransform;
    }
}
```

(4) 定义模拟雾效时使用的各个参数:

```
[Range(0.1f, 3.0f)]
public float fogDensity = 1.0f;

public Color fogColor = Color.white;

public float fogStart = 0.0f;
public float fogEnd = 2.0f;

public Texture noiseTexture;

[Range(-0.5f, 0.5f)]
public float fogXSpeed = 0.1f;

[Range(-0.5f, 0.5f)]
public float fogYSpeed = 0.1f;

[Range(0.0f, 3.0f)]
public float noiseAmount = 1.0f;
```

**fogDensity**用于控制雾的浓度，**fogColor**用于控制雾的颜色。我们使用的雾效模拟函数是基于高度的，因此参数**fogStart**用于控制雾效的起始高度，**fogEnd**用于控制雾效的终止高度。**noiseTexture**是我们使用的噪声纹理，**fogXSpeed**和**fogYSpeed**分别对应了噪声纹理在X和Y方向上的移动速度，以此来模拟雾的飘动效果。最后，**noiseAmount**用于控制噪声程度，当**noiseAmount**为0时，表示不应用任何噪声，即得到一个均匀的基于高度的全局雾效。

(5) 由于本例需要获取摄像机的深度纹理，我们在脚本的 `OnEnable` 函数中设置摄像机的相应状态：

```
void OnEnable() {  
    camera.depthTextureMode |= DepthTextureMode.Depth;  
}
```

(6) 最后，我们实现了 `OnRenderImage` 函数：

```
void OnRenderImage (RenderTexture src, RenderTexture dest) {  
    if (material != null) {  
        Matrix4x4 frustumCorners = Matrix4x4.identity;  
  
        // Compute frustumCorners  
        ...  
  
        material.SetMatrix("_FrustumCornersRay", frustumCorners);  
  
        material.SetFloat("_FogDensity", fogDensity);  
        material.SetColor("_FogColor", fogColor);  
        material.SetFloat("_FogStart", fogStart);  
        material.SetFloat("_FogEnd", fogEnd);  
  
        material.SetTexture("_NoiseTex", noiseTexture);  
        material.SetFloat("_FogXSpeed", fogXSpeed);  
        material.SetFloat("_FogYSpeed", fogYSpeed);  
        material.SetFloat("_NoiseAmount", noiseAmount);  
  
        Graphics.Blit (src, dest, material);  
    } else {  
        Graphics.Blit(src, dest);  
    }  
}
```

我们首先利用13.3节学习的方法计算近裁剪平面的4个角对应的向量，并把它们存储在一个矩阵类型的变量（`frustumCorners`）中。计算过程和原理均可参见13.3节。随后，我们把结果和其他参数传递给材质，并调用 `Graphics.Blit (src, dest, material)` 把渲染结果显示在屏幕上。

下面，我们来实现Shader的部分。打开Chapter15-FogWithNoise，进行如下修改。

(1) 我们首先需要声明本例使用的各个属性：

```
Properties {
    _MainTex ("Base (RGB)", 2D) = "white" {}
    _FogDensity ("Fog Density", Float) = 1.0
    _FogColor ("Fog Color", Color) = (1, 1, 1, 1)
    _FogStart ("Fog Start", Float) = 0.0
    _FogEnd ("Fog End", Float) = 1.0
    _NoiseTex ("Noise Texture", 2D) = "white" {}
    _FogXSpeed ("Fog Horizontal Speed", Float) = 0.1
    _FogYSpeed ("Fog Vertical Speed", Float) = 0.1
    _NoiseAmount ("Noise Amount", Float) = 1
}
```

(2) 在本节中，我们使用CGINCLUDE来组织代码。我们在SubShader块中利用CGINCLUDE和ENDCG语义来定义一系列代码：

```
SubShader {
    CGINCLUDE
    ...
    ENDCG
    ...
}
```

(3) 声明代码中需要使用的各个变量：

```
float4x4 _FrustumCornersRay;

sampler2D _MainTex;
half4 _MainTex_TexelSize;
sampler2D _CameraDepthTexture;
half _FogDensity;
fixed4 _FogColor;
float _FogStart;
float _FogEnd;
sampler2D _NoiseTex;
half _FogXSpeed;
half _FogYSpeed;
half _NoiseAmount;
```



`_FrustumCornersRay`虽然没有在**Properties**中声明，但仍可由脚本传递给**Shader**。除了**Properties**中声明的各个属性，我们还声明了深度纹理 `_CameraDepthTexture`，Unity会在背后把得到的深度纹理传递给该值。

(4) 定义顶点着色器，这和13.3节中的实现完全一致。读者可以在13.3节找到它的实现和相关解释。

(5) 定义片元着色器：

```
fixed4 frag(v2f i) : SV_Target {
    float linearDepth =
    LinearEyeDepth(SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, i.
    uv_depth));
    float3 worldPos = _WorldSpaceCameraPos + linearDepth *
    i.interpolatedRay.xyz;

    float2 speed = _Time.y * float2(_FogXSpeed, _FogYSpeed);
    float noise = (tex2D(_NoiseTex, i.uv + speed).r - 0.5) *
    _NoiseAmount;

    float fogDensity = (_FogEnd - worldPos.y) / (_FogEnd -
    _FogStart);
    fogDensity = saturate(fogDensity * _FogDensity * (1 + noise));

    fixed4 finalColor = tex2D(_MainTex, i.uv);
    finalColor.rgb = lerp(finalColor.rgb, _FogColor.rgb,
    fogDensity);

    return finalColor;
}
```

我们首先根据深度纹理来重建该像素在世界空间中的位置。然后，我们利用内置的`_Time.y`变量和`_FogXSpeed`、`_FogYSpeed`属性计算出当前噪声纹理的偏移量，并据此对噪声纹理进行采样，得到噪声值。我们把该值减去0.5，再乘以控制噪声程度的属性`_NoiseAmount`，得到最终的噪声值。随后，我们把该噪声值添加到雾效浓度的计算

中，得到应用噪声后的雾效混合系数`fogDensity`。最后，我们使用该系数将雾的颜色和原始颜色进行混合后返回。

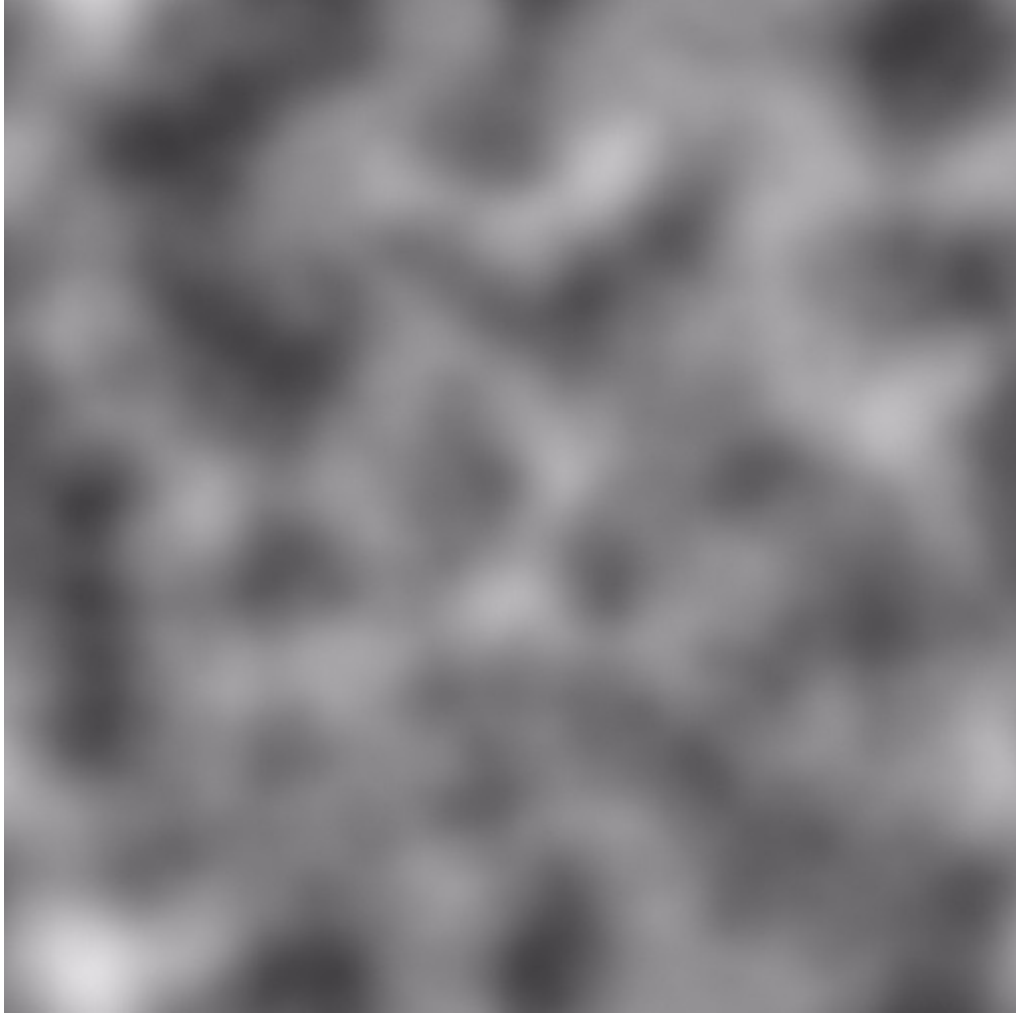
(6) 随后，我们定义了雾效渲染所需的`Pass`:

```
Pass {  
    CGPROGRAM  
  
    #pragma vertex vert  
    #pragma fragment frag  
  
    ENDCG  
}
```

(7) 最后，我们关闭了`Shader`的`Fallback`:

```
Fallback Off
```

完成后返回编辑器，并把`Chapter15-FogWithNoise`拖曳到摄像机的`FogWithNoise.cs`脚本中的`fogShader`参数中。当然，我们可以在`FogWithNoise.cs`的脚本面板中将`fogShader`参数的默认值设置为`Chapter15-FogWithNoise`，这样就不需要以后使用时每次都手动拖曳了。本节使用的噪声纹理（对应本书资源的`Assets/Textures/Chapter15/Fog_Noise.jpg`）如图15.7所示。我们把该噪声纹理拖曳到`FogWithNoise.cs`脚本中的`noiseTexture`参数中，我们也可以参照之前的方法，直接在`FogWithNoise.cs`的脚本面板中将`noiseTexture`参数的默认值设置为`Fog_Noise.jpg`，这样就不需要以后使用时每次都手动拖曳了。



▲图15.7 本节使用的噪声纹理

## 15.4 扩展阅读

读者在阅读本章时，可能会有一个疑问：这些噪声纹理都是如何构建出来的？这些噪声纹理可以被认为是一种程序纹理（**Procedure Texture**），它们都是由计算机利用某些算法生成的。Perlin噪声（[https://en.wikipedia.org/wiki/Perlin\\_noise](https://en.wikipedia.org/wiki/Perlin_noise)）和Worley噪声（[https://en.wikipedia.org/wiki/Worley\\_noise](https://en.wikipedia.org/wiki/Worley_noise)）是两种最常使用的噪声类型，例如我们在15.3节中使用的噪声纹理由Perlin噪声生成而来。Perlin

噪声可以用于生成更自然的噪声纹理，而Worley噪声则通常用于模拟诸如石头、水、纸张等多孔噪声。现代的图像编辑软件，如Photoshop等，往往提供了类似的功能或插件，以帮助美术人员生成需要的噪声纹理，但如果读者想要更加自由地控制噪声纹理的生成，可能就需要了解它们的生成原理。读者可以在这个博客

(<http://flafla2.github.io/2014/08/09/perlinnoise.html>) 中找到一篇关于理解Perlin噪声的非常好的文章，在文章的最后，作者还给出了很多其他出色的参考链接。关于Worley噪声，读者可以在作者Worley1998年发表的论文<sup>[1]</sup>中找到它的算法和实现细节。在另一个非常好的博客

(<http://scrawkblog.com/category/procedural-noise/>) 中，博主给出了很多程序噪声在Unity中的实现，并包含了实现源码。

## 15.5 参考文献

[1] Worley S. A cellular texture basis function[C]//Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. ACM, 1996: 291-294。

## 第16章 Unity中的渲染优化技术

程序优化的第一条准则：不要优化。程序优化的第二条准则（仅针对专家！）：不要优化。

——Michael A. Jackson

在进行程序优化的时候，人们经常会引用英国的计算机科学家Michael A. Jackson在1988年的优化准则。Jackson是想借此强调，对问题认识不清以及过度优化往往会让事情变得更加复杂，产生更多的程序错误。

然而，如果我们在游戏开发过程中从来都没有考虑优化，那么结果往往是惨不忍睹的。一个正确的做法是，从一开始就把优化当成是游戏设计中的一部分。正在阅读本书的读者，有可能是移动游戏的开发者。和PC相比，移动设备上的GPU有着完全不同的架构设计，它能使用的带宽、功能和其他资源都非常有限。这要求我们需要时刻把优化谨记在心，才可以避免等到项目完成时才发现游戏根本无法在移动设备上流畅运行的结果。

在本章，我们将会阐述一些Unity中常见的优化技术。这些优化技术都是和渲染相关的，例如，使用批处理、LOD（Level of Detail）技术等。在本章最后的扩展阅读部分，我们给出一些非常有价值的参考资料，在那里读者可以学习到更多真实项目中的优化技术。

在开始学习之前，我们希望读者能够理解，游戏优化不仅是程序员的工作，更需要美工人员在游戏的美术上进行一定的权衡，例如，避免使用全屏的屏幕特效，避免使用计算复杂的shader，减少透明混合造成的overdraw等。也就是说，这是由程序员和美工人员等各个部分人员共同参与的工作。

## 16.1 移动平台的特点

和PC平台相比，移动平台上的GPU架构有很大的不同。由于处理资源等条件的限制，移动设备上的GPU架构专注于尽可能使用更小的带宽和功能，也由此带来了许多和PC平台完全不同的现象。

例如，为了尽可能移除那些隐藏的表面，减少overdraw（即一个像素被绘制多次），PowerVR芯片（通常用于iOS设备和某些Android设备）使用了**基于瓦片的延迟渲染（Tiled-based Deferred Rendering, TBDR）**架构，把所有的渲染图像装入一个个瓦片（tile）中，再由硬件找到可见的片元，而只有这些可见片元才会执行片元着色器。另一些基于瓦片的GPU架构，如Adreno（高通的芯片）和Mali（ARM的芯片）则会使用Early-Z或相似的技术进行一个低精度的深度检测，来剔除那些不需要渲染的片元。还有一些GPU，如Tegra（英伟达的芯片），则使用了传统的架构设计，因此在这些设备上，overdraw更可能造成性能的瓶颈。

由于这些芯片架构造成的不同，一些游戏往往需要针对不同的芯片发布不同的版本，以便对每个芯片进行更有针对性的优化。尤其是在Android平台上，不同设备使用的硬件，如图形芯片、屏幕分辨率

等，大相径庭，这对图形优化提出了更高的挑战。相比与Android平台，iOS平台的硬件条件则相对统一。读者可以在Unity手册的**iOS硬件指南**（<http://docs.unity3d.com/Manual/iphone-Hardware.html>）中找到相关的资料。

## 16.2 影响性能的因素

首先，在学习如何优化之前，我们得了解影响游戏性能的因素有哪些，才能对症下药。对于一个游戏来说，它主要需要使用两种计算资源：**CPU**和**GPU**。它们会互相合作，来让我们的游戏可以在预期的**帧率**和**分辨率**下工作。其中，**CPU**主要负责保证帧率，**GPU**主要负责分辨率相关的一些处理。

据此，我们可以把造成游戏性能瓶颈的主要原因分成以下几个方面。

### (1) CPU。

- 过多的draw call。
- 复杂的脚本或者物理模拟。

### (2) GPU。

- 顶点处理。
  - 过多的顶点。
  - 过多的逐顶点计算。
- 片元处理。



- 过多的片元（既可能是由于分辨率造成的，也可能是由于 **overdraw** 造成的）。
- 过多的逐片元计算。

### (3) 带宽。

- 使用了尺寸很大且未压缩的纹理。
- 分辨率过高的帧缓存。

对于CPU来说，限制它的主要是每一帧中draw call的数目。我们曾在2.2节和2.4.3节中介绍过draw call的相关概念和原理。简单来说，就是CPU在每次通知GPU进行渲染之前，都需要提前准备好顶点数据（如位置、法线、颜色、纹理坐标等），然后调用一系列API把它们放到GPU可以访问到的指定位置，最后，调用一个绘制命令，来告诉GPU，“嘿，我把东西都准备好了，你赶紧出来干活（渲染）吧！”。而调用绘制命令的时候，就会产生一个draw call。过多的draw call会造成CPU的性能瓶颈，这是因为每次调用draw call时，CPU往往都需要改变很多渲染状态的设置，而这些操作是非常耗时的。如果一帧中需要的draw call数目过多的话，就会导致CPU把大部分时间都花费在提交draw call的工作上面了。当然，其他原因也可能造成CPU瓶颈，例如物理、布料模拟、蒙皮、粒子模拟等，这些都是计算量很大的操作，但由于本书主要讨论Shader方面的相关技术，因此，这些内容不在本书的讨论范围内。

而对于GPU来说，它负责整个渲染流水线。它从处理CPU传递过来的模型数据开始，进行顶点着色器、片元着色器等一系列工作，最后输出屏幕上的每个像素。因此，GPU的性能瓶颈和需要处理的顶点

数目、屏幕分辨率、显存等因素有关。而相关的优化策略可以从减少处理的数据规模（包括顶点数目和片元数目）、减少运算复杂度等方面入手。

在了解了上面基本的内容后，本章后续章节会涉及的优化技术有。

### （1）CPU优化。

- 使用批处理技术减少draw call数目。

### （2）GPU优化。

- 减少需要处理的顶点数目。
  - 优化几何体。
  - 使用模型的LOD（Level of Detail）技术。
  - 使用遮挡剔除（Occlusion Culling）技术。
- 减少需要处理的片元数目。
  - 控制绘制顺序。
  - 警惕透明物体。
  - 减少实时光照。
- 减少计算复杂度。
  - 使用Shader的LOD（Level of Detail）技术。
  - 代码方面的优化。

### （3）节省内存带宽。

- 减少纹理大小。

- 利用分辨率缩放。

在开始优化之前，我们首先需要知道是哪个步骤造成了性能瓶颈。而这可以利用Unity提供的一些渲染分析工具来实现。

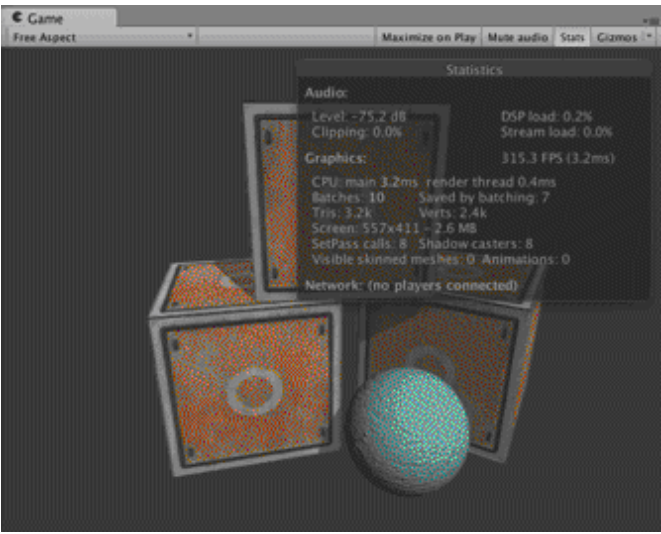
## 16.3 Unity中的渲染分析工具

Unity内置了一些工具，来帮助我们方便地查看和渲染相关的各个统计数据。这些数据可以帮助我们分析游戏渲染性能，从而更有针对性地进行优化。在Unity 5中，这些工具包括了渲染统计窗口

（Rendering Statistics Window）、性能分析器（Profiler），以及帧调试器（Frame Debugger）。需要注意的是，在不同的目标平台上，这些工具中显示的数据也会发生变化。

### 16.3.1 认识Unity 5的渲染统计窗口

Unity 5提供了一个全新的窗口，即**渲染统计窗口（Rendering Statistics Window）**来显示当前游戏的各个渲染统计变量，我们可以通过在Game视图右上方的菜单中单击Stats按钮来打开它，如图16.1所示。从图16.1中可以看出，渲染统计窗口主要包含了3个方面的信息：音频（Audio）、图像（Graphics）和网络（Network）。我们这里只关注第二个方面，即图像相关的渲染统计结果。



▲图16.1 Unity 5的渲染统计窗口

渲染统计窗口中显示了很多重要的渲染数据，例如FPS、批处理数目、顶点和三角网格的数目等。表16.1列出了渲染统计窗口中显示各个信息。

表16.1

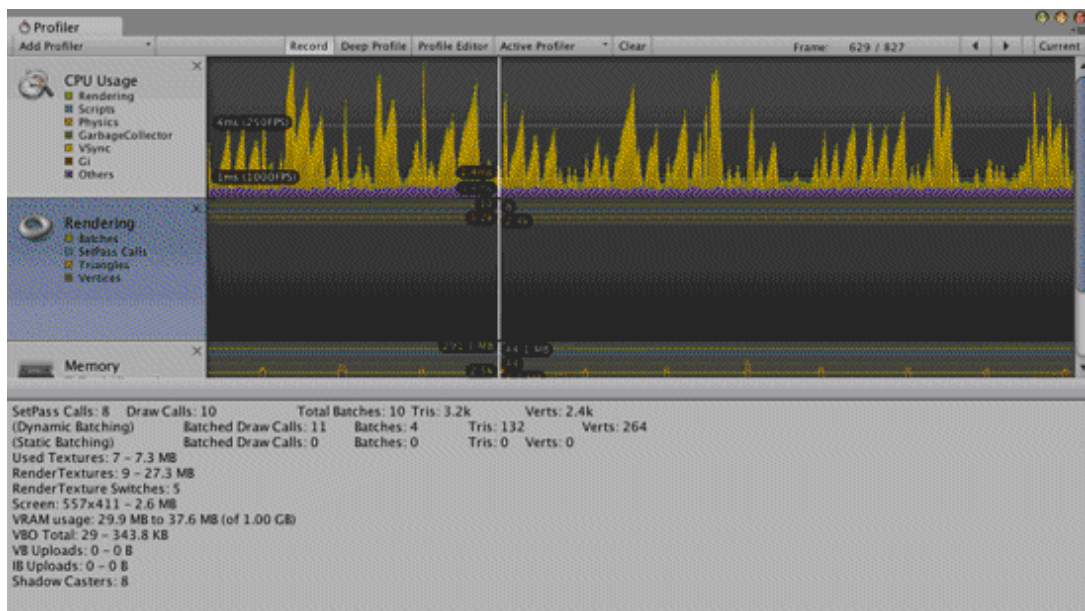
信息名称	描 述
每帧的时间和FPS	在 <b>Graphic</b> 的右侧显示，给出了处理和渲染一帧所需的时间，以及FPS数目
Batches	一帧中需要进行的批处理数目
Saved by batching	合并的批处理数目，这个数字表明了批处理为我们节省了多少draw call
Tris和Verts	需要绘制的三角面片和顶点数目

信息名称	描 述
Screen	屏幕的大小，以及它占用的内存大小
SetPass	渲染使用的Pass的数目，每个Pass都需要Unity的runtime来绑定一个新的Shader，这可能造成CPU的瓶颈
Visible Skinned Meshes	渲染的蒙皮网格的数目
Animations	播放的动画数目

Unity 5的渲染统计窗口相较于之前版本中的有了一些变化，最明显的区别之一就是去掉了draw call数目的显示，而添加了批处理数目的显示。Batches和Saved by batching更容易让开发者理解批处理的优化结果。当然，如果我们想要查看draw call的数目等其他更加详细的数据，可以通过Unity编辑器的性能分析器来查看。

### 16.3.2 性能分析器的渲染区域

我们可以通过单击Window -> Profiler来打开Unity的**性能分析器（Profiler）**。性能分析器中的渲染区域（Rendering Area）提供了更多关于渲染的统计信息，图16.2给出了对图16.1中场景的渲染分析结果。



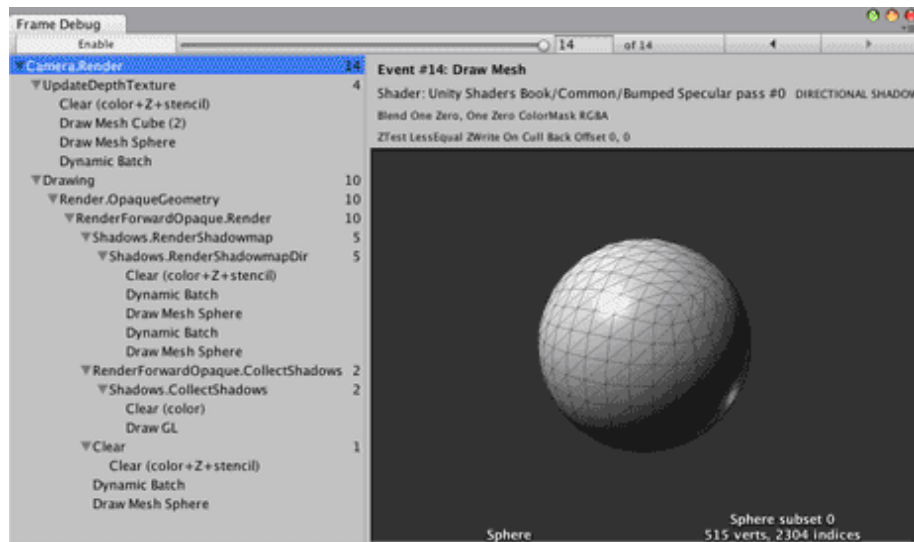
▲ 图16.2 使用Unity的性能分析器中的渲染区域来查看更多关于渲染的统计信息

性能分析器显示了绝大部分在渲染统计窗口中提供的信息，例如，绿线显示了批处理数目、蓝线显示了Pass数目等，同时还给出了许多其他非常有用的信息，例如，draw call数目、动态批处理/静态批处理的数目、渲染纹理的数目和内存占用等。

结合渲染统计窗口和性能分析器，我们可以查看与渲染相关的绝大多数重要的数据。一个值得注意的现象是，性能分析器给出的draw call数目和批处理数目、Pass数目并不相等，并且看起来好像要大于我们估算的数目，这是因为Unity在背后需要进行很多工作，例如，初始化各个缓存、为阴影更新深度纹理和阴影映射纹理等，因此需要花费比“预期”更多的draw call。一个好消息是，Unity 5引入了一个新的工具来帮助我们查看每一个draw call的工作，这个工具就是帧调试器。

### 16.3.3 再谈帧调试器

我们已经在之前的章节中多次看到**帧调试器（Frame Debugger）**的应用，例如5.5.3节中解释了如何使用帧调试器来对Shader进行调试。我们可以通过Window -> Frame Debugger来打开它。在这个窗口中，我们可以清楚地看到每一个draw call的工作和结果，如图16.3所示。



▲ 图16.3 使用帧调试器来查看单独的draw call的绘制结果

帧调试器的调试面板上显示了渲染这一帧所需要的所有的渲染事件，在本例中，事件数目为14，而其中包含了10个draw call事件（其他渲染事件多为清空缓存等）。通过单击面板上的每个事件，我们可以在Game视图查看该事件的绘制结果，同时渲染统计面板上的数据也会显示成截止到当前事件为止的各个渲染统计数据。以本例为例（场景如图16.1所示），要渲染一帧共需要花费10个draw call，其中4个draw call用于更新深度纹理（对应UpdateDepthTexture），4个draw call用于渲染平行光的阴影映射纹理，1个draw call用于绘制动态批处理后的3个立方体模型，1个draw call用于绘制球体。



在Unity的渲染统计窗口、分析器和帧调试器这3个利器的帮助下，我们可以获得很多有用的优化信息。但是，很多诸如渲染时间这样的数据是基于当前的开发平台得到的，而非真机上的结果。事实上，Unity正在和硬件生产商合作，来首先让使用英伟达图睿（Tegra）的设备可以出现在Unity的性能分析器中。我们有理由相信，在后续的Unity版本中，直接在Unity中对移动设备进行性能分析不再是梦想。然而，在这个梦想实现之前，我们仍然需要一些外部的性能分析工具的帮助。

### 16.3.4 其他性能分析工具

对于移动平台上的游戏来说，我们更希望得到在真机上运行游戏时的性能数据。这时，Unity目前提供的各个工具可能就不再能满足我们的需求了。

对于Android平台来说，高通的Adreno分析工具可以对不同的测试机进行详细的性能分析。英伟达提供了NVPerfHUD工具来帮助我们得到几乎所有需要的性能分析数据，例如，每个draw call的GPU时间，每个shader花费的cycle数目等。

对于iOS平台来说，Unity内置的分析器可以得到整个场景花费的GPU时间。PowerVRAM的PVRUniSCo shader分析器也可以给出一个大致的性能评估。Xcode中的OpenGL ES Driver Instruments可以给出一些宏观上的性能信息，例如，设备利用率、渲染器利用率等。但相对于Android平台，对iOS的性能分析更加困难（工具较少）。而且PowerVR芯片采用了基于瓦片的延迟渲染器，因此，想要得到每个draw call花费

的GPU时间是几乎不可能的。这时，一些宏观上的统计数据可能更有参考价值。

一些其他的性能分析工具可以在Unity的官方手册（<http://docs.unity3d.com/Manual/MobileProfiling.html>）中找到。当找到了性能瓶颈后，我们就可以针对这些方面进行特定的优化。

## 16.4 减少draw call数目

读者最常看到的优化技术大概就是**批处理（batching）**了。批处理的实现原理就是为了减少每一帧需要的draw call数目。为了把一个对象渲染到屏幕上，CPU需要检查哪些光源影响了该物体，绑定shader并设置它的参数，再把渲染命令发送给GPU。当场景中包含了大量对象时，这些操作就会非常耗时。一个极端的例子是，如果我们需要渲染一千个三角形，把它们按一千个单独的网格进行渲染所花费的时间要远远大于渲染一个包含了一千个三角形的网格。在这两种情况下，GPU的性能消耗其实并没有多大的区别，但CPU的draw call数目就会成为性能瓶颈。因此，批处理的思想很简单，就是在每次调用draw call时尽可能多地处理多个物体。我们已经在2.2节和2.4.3节中详细地讲述了draw call和批处理之间的联系，本节旨在介绍如何在Unity中利用批处理技术来优化渲染。

那么，什么样的物体可以一起处理呢？答案就是使用同一个材质的物体。这是因为，对于使用同一个材质的物体，它们之间的不同仅仅在于顶点数据的差别。我们可以把这些顶点数据合并在一起，再一起发送给GPU，就可以完成一次批处理。

Unity中支持两种批处理方式：一种是动态批处理，另一种是静态批处理。对于动态批处理来说，优点是一切处理都是Unity自动完成的，不需要我们自己做任何操作，而且物体是可以移动的，但缺点是，限制很多，可能一不小心就会破坏了这种机制，导致Unity无法动态批处理一些使用了相同材质的物体。而对于静态批处理来说，它的优点是自由度很高，限制很少；但缺点是可能会占用更多的内存，而且经过静态批处理后的所有物体都不可以再移动了（即便在脚本中尝试改变物体的位置也是无效的）。

### 16.4.1 动态批处理

如果场景中有一些模型共享了同一个材质并满足一些条件，Unity就可以自动把它们进行批处理，从而只需要花费一个draw call就可以渲染所有的模型。动态批处理的基本原理是，每一帧把可以进行批处理的模型网格进行合并，再把合并后模型数据传递给GPU，然后使用同一个材质对其渲染。除了实现方便，动态批处理的另一个好处是，经过批处理的物体仍然可以移动，这是由于在处理每帧时Unity都会重新合并一次网格。

虽然Unity的动态批处理不需要我们进行任何额外工作，但只有满足条件的模型和材质才可以被动态批处理。需要注意的是，随着Unity版本的变化，这些条件也有一些改变。在本节中，我们给出一些主要的条件限制。

- 能够进行动态批处理的网格的顶点属性规模要小于900。例如，如果shader中需要使用顶点位置、法线和纹理坐标这3个顶点属性，那么要想让模型能够被动态批处理，它的顶点数目不能超过300。

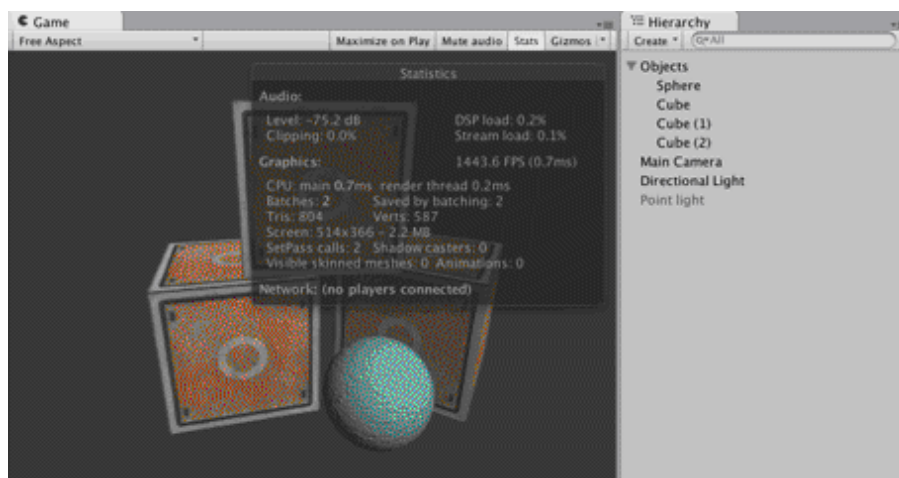
需要注意的是，这个数字在未来有可能会发生变化，因此不要依赖这个数据。

- 一般来说，所有对象都需要使用同一个缩放尺度（可以是(1, 1, 1)、(1, 2, 3)、(1.5, 1.4, 1.3)等，但必须都一样）。一个例外情况是，如果所有的物体都使用了不同的非统一缩放，那么它们也是可以被动态批处理的。但在Unity 5中，这种对模型缩放的限制已经不存在了。
- 使用光照纹理（lightmap）的物体需要小心处理。这些物体需要额外的渲染参数，例如，在光照纹理上的索引、偏移量和缩放信息等。因此，为了让这些物体可以被动态批处理，我们需要保证它们指向光照纹理中的同一个位置。
- 多Pass的shader会中断批处理。在前向渲染中，我们有时需要使用额外的Pass来为模型添加更多的光照效果，但这样一来模型就不会被动态批处理了。

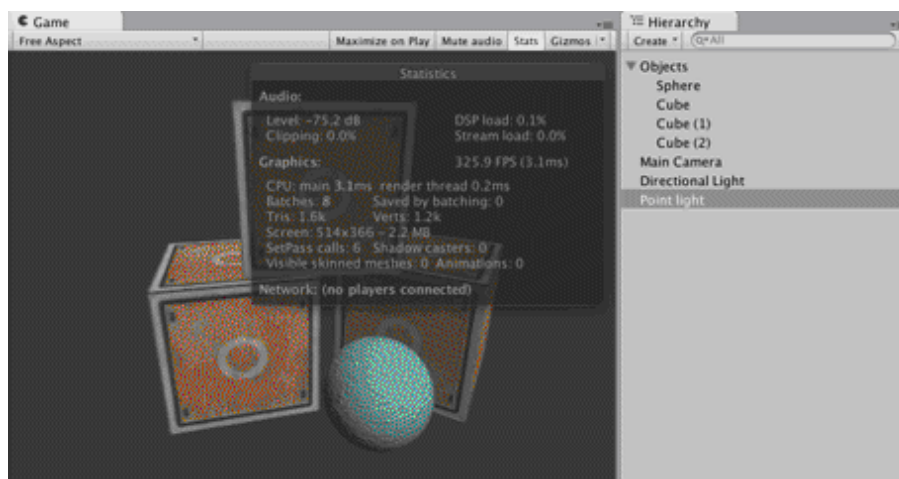
在本书资源的Scene\_16\_3\_1场景中，我们给出了这样一个场景。场景中包含了3个立方体，它们使用同一个材质，同时还包含了一个使用其他材质的球体。场景中还包括了一个平行光，但我们关闭了它的阴影效果，以避免阴影计算对批处理数目的影响。这样一个场景的渲染统计数据如图16.4所示。

从图16.4中可以看出，要渲染这样一个包含了4个物体的场景共需要两个批处理。其中，一个批处理用于绘制经过动态批处理合并后的3个立方体网格，另一个批处理用于绘制球体。我们可以从**Save by batching**看出批处理帮我们节省了两个draw call。

现在，我们再向场景中添加一个点光源，并调整它的位置使它可以照亮场景中的4个物体。由于场景中的物体都使用了多个Pass的shader，因此，点光源会对它们产生光照影响。图16.5给出了添加点光源后的渲染统计数据。



▲ 图16.4 动态批处理



▲ 图16.5 多光源对动态批处理的影响结果

从图16.5中可以看出，渲染一帧所需的批处理数目增大到了8，而**Save by batching**的数目也变成了0。这是因为，使用了多个Pass的

shader在需要应用多个光照的情况下，破坏了动态批处理的机制，导致Unity不能对这些物体进行动态批处理。而由于平行光和点光源需要对4个物体分别产生影响，因此，需要 $2 \times 4$ 个批处理操作。需要注意的是，只有物体在点光源的影响范围内，Unity才会调用额外的Pass来处理它。因此，如果场景中点光源距离物体很远，那么它们仍然会被动态批处理的。

动态批处理的限制条件比较多，例如很多时候，我们的模型数据往往会超过900的顶点属性限制。这种时候依赖动态批处理来减少draw call显然已经不能够满足我们的需求了。这时，我们可以使用Unity的静态批处理技术。

### 16.4.2 静态批处理

Unity提供了另一种批处理方式，即静态批处理。相比于动态批处理来说，静态批处理适用于任何大小的几何模型。它的实现原理是，只在运行开始阶段，把需要进行静态批处理的模型合并到一个新的网格结构中，这意味着这些模型不可以在运行时刻被移动。但由于它只需要进行一次合并操作，因此，比动态批处理更加高效。静态批处理的另一个缺点在于，它往往需要占用更多的内存来存储合并后的几何结构。这是因为，如果在静态批处理前一些物体共享了相同的网格，那么在内存中每一个物体都会对应一个该网格的复制品，即一个网格会变成多个网格再发送给GPU。如果这类使用同一网格的对象很多，那么这就会成为一个性能瓶颈了。例如，如果在一个使用了1 000个相同树模型的森林中使用静态批处理，那么，就会多使用1 000倍的内存，这会造成严重的内存影响。这种时候，解决方法要么忍受这种牺牲内存换取性能的方法，要么不要使用静态批处理，而使用动态批处

理技术（但要小心控制模型的顶点属性数目），或者自己编写批处理的方法。

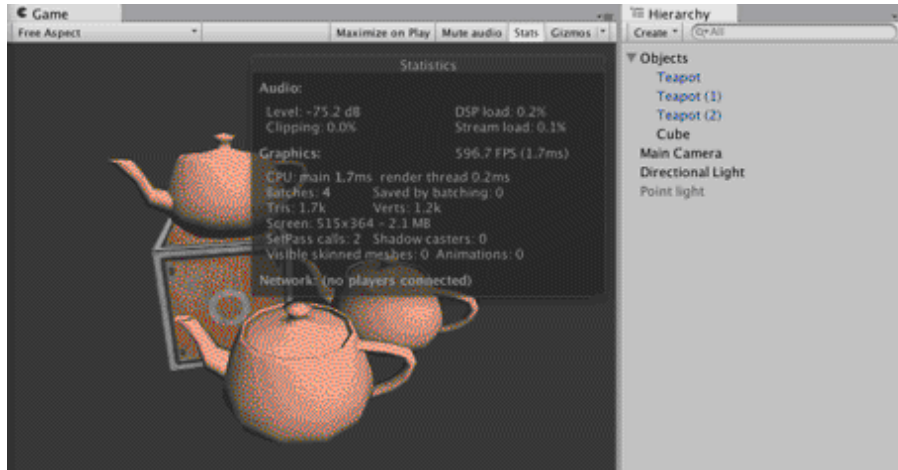
在本书资源的Scene\_16\_3\_2场景中，我们给出了一个测试静态批处理的场景。场景中包含了3个Teapot模型，它们使用同一个材质，同时还包含了一个使用不同材质的立方体。场景中还包含了一个平行光，但我们关闭了它的阴影效果，以避免阴影计算对批处理数目的影响。在运行前，这样一个场景的渲染统计数据如图16.6所示。

从图16.6中可以看出，尽管3个Teapot模型使用了相同的材质，但它们仍然没有被动态批处理。这是因为，Teapot模型包含的顶点数目是393，而它们使用的shader中需要使用4个顶点属性（顶点位置、法线方向、切线方向和纹理坐标），超过了动态批处理中限定的900限制。此时，要想减少draw call就需要使用静态批处理。

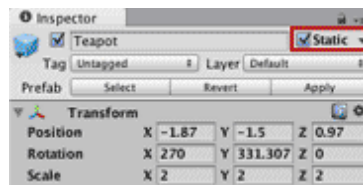
静态批处理的实现非常简单，只需要把物体面板上的**Static**复选框勾选上即可（实际上我们只需要勾选Batching Static即可），如图16.7所示。

这时，我们再观察渲染统计窗口中的批处理数目，还是没有变化。但是不要急，运行程序后，变化就出现了，如图16.8所示。



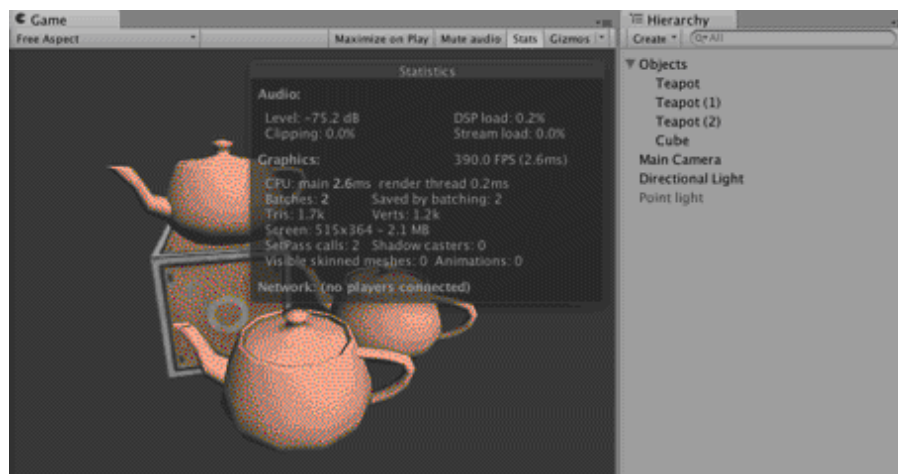


▲ 图16.6 静态批处理前的渲染统计数据

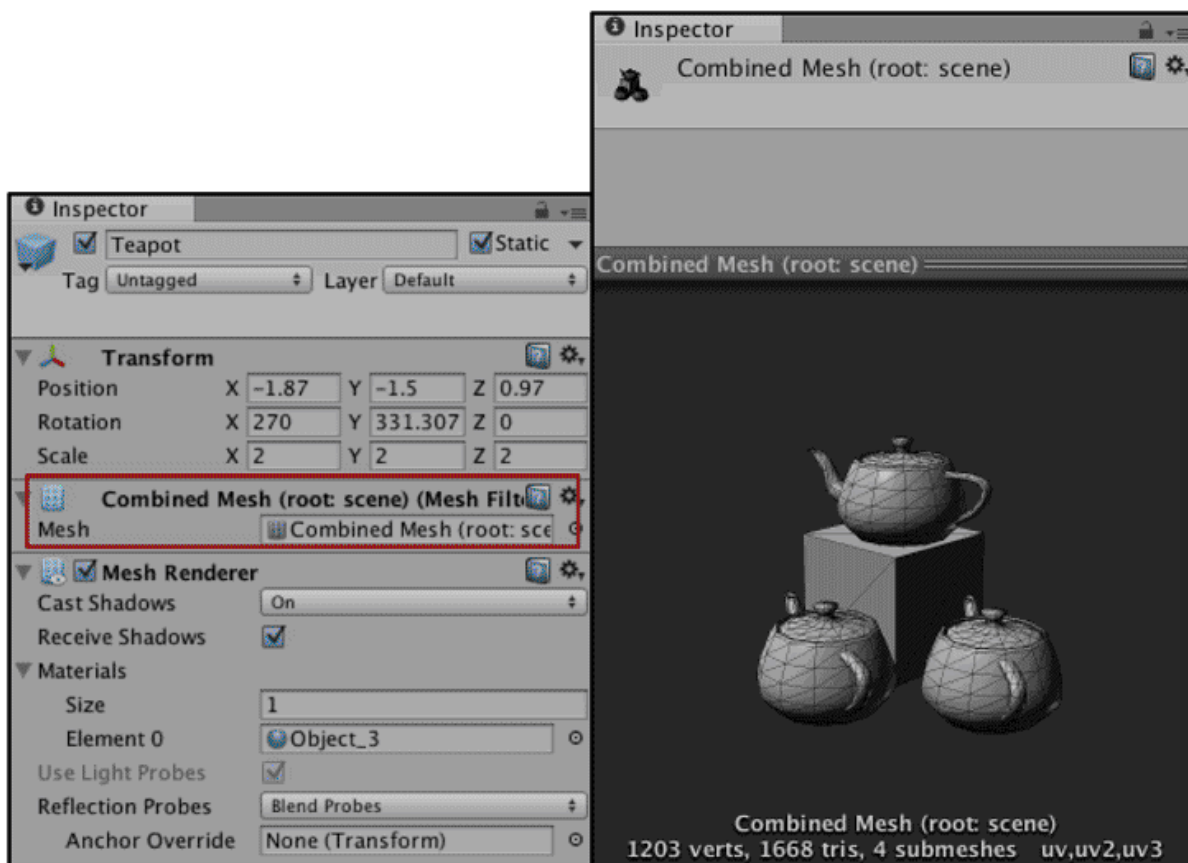


▲ 图16.7 把物体标志为Static

从图16.2中可以看出，现在的批处理数目变成了2，而**Save by batching**数目也显示为2。此时，如果我们在运行时查看每个模型使用的网格，会发现它们都变成了一个名为**Combined Mesh (root: scene)**的东西，如图16.9所示。这个网格是Unity合并了所有被标识为“Static”的物体的结果，在我们的例子里，就是3个**Teapot**和一个立方体。读者可能会有一个疑问，这4个对象明明不是都使用了一个材质，为什么可以合并成一个呢？如果你仔细观看图16.9的结果，会发现在图16.9的右下方标明了“4 submeshes”，也就是说，这个合并后的网格其实包含了4个子网格，即场景中的4个对象。对于合并后的网格，Unity会判断其中使用同一个材质的子网格，然后对它们进行批处理。

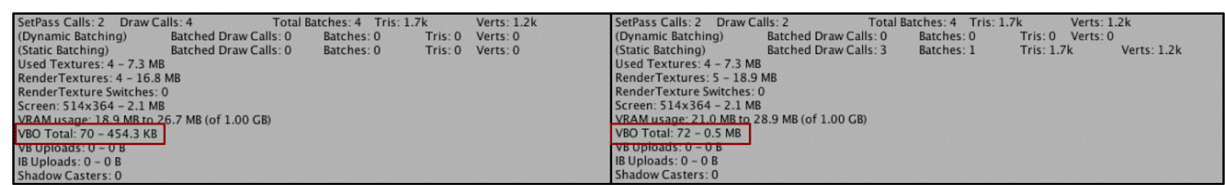


▲ 图16.8 静态批处理



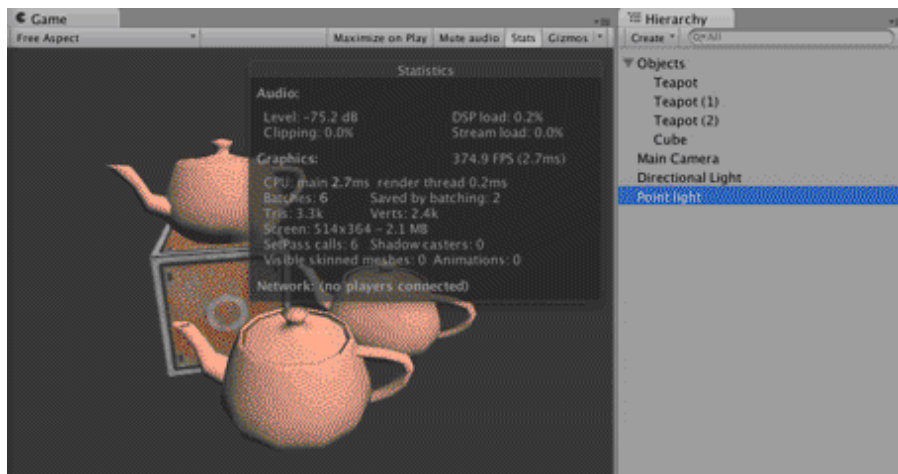
▲ 图16.9 静态批处理中Unity会合并所有被标识为“Static”的物体

在内部实现上，Unity首先把这些静态物体变换到世界空间下，然后为它们构建一个更大的顶点和索引缓存。对于使用了同一材质的物体，Unity只需要调用一个draw call就可以绘制全部物体。而对于使用了不同材质的物体，静态批处理同样可以提升渲染性能。尽管这些物体仍然需要调用多个draw call，但静态批处理可以减少这些draw call之间的状态切换，而这些切换往往是费时的操作。从合并后的网格结构中我们还可以发现，尽管3个Teapot对象使用了同一个网格，但合并后却变成了3个独立网格。而且，我们可以从Unity的分析器中观察到在应用静态批处理前后VBO total的变化，从图16.10所示中可以看出，VBO（Vertex Buffer Object，顶点缓冲对象）的数目变大了。这正是因为静态批处理会占用更多内存的缘故，正如本节一开头所讲，静态批处理需要占用更多的内存来存储合并后的几何结构，如果一些物体共享了相同的网格，那么在内存中每一个物体都会对应一个该网格的复制品。



▲ 图16.10 静态批处理会占用更多的内存。左边：静态批处理前的渲染统计数据，右边：静态批处理后的渲染统计数据

如果场景中包含了除了平行光以外的其他光源，并且在shader中定义了额外的Pass来处理它们，这些额外的Pass部分是不会被批处理的。图16.11显示了在场景中添加了一个会影响4个物体的点光源之后，渲染统计窗口的数据变化。



▲ 图16.11 处理其他逐像素光的Pass不会被静态批处理

但是，处理平行光的Base Pass部分仍然会被静态批处理，因此，我们仍然可以节省两个draw call。

### 16.4.3 共享材质

从之前的内容可以看出，无论是动态批处理还是静态批处理，都要求模型之间需要共享同一个材质。但不同的模型之间总会需要有不同的渲染属性，例如，使用不同的纹理、颜色等。这时，我们需要一些策略来尽可能地合并材质。

如果两个材质之间只有使用的纹理不同，我们可以把这些纹理合并到一张更大的纹理中，这张更大的纹理被称为是一张图集（atlas）。一旦使用了同一张纹理，我们就可以使用同一个材质，再使用不同的采样坐标对纹理采样即可。

但有时，除了纹理不同外，不同的物体在材质上还有一些微小的参数变化，例如，颜色不同、某些浮点属性不同。但是，不管是动态批处理还是静态批处理，它们的前提都是要使用同一个材质。是同一

个，而不是使用了同一种Shader的材质，也就是说它们指向的材质必须是同一个实体。这意味着，只要我们调整了参数，就会影响到所有使用这个材质的对象。那么想要微小的调整怎么办呢？一种常用的方法就是使用网格的顶点数据（最常见的就是顶点颜色数据）来存储这些参数。

前面说过，经过批处理后的物体会被处理成更大的VBO发送给GPU，VBO中的数据可以作为输入传递给顶点着色器，因此，我们可以巧妙地控制VBO中的数据，从而达到不同效果的目的。一个例子是，森林场景中所有的树使用了同一种材质，我们希望它们可以通过批处理来减少draw call，但不同树的颜色可能不同。这时，我们可以利用网格的顶点的颜色数据来调整。

需要注意的是，如果我们需要在脚本中访问共享材质，应该使用Renderer.sharedMaterial来保证修改的是和其他物体共享的材质，但这意味着修改会应用到所有使用该材质的物体上。另一个类似的API是Renderer.material，如果使用Renderer.material来修改材质，Unity会创建一个该材质的复制品，从而破坏批处理在该物体上的应用，这可能并不是我们希望看到的。

#### 16.4.4 批处理的注意事项

在选择使用动态批处理还是静态批处理时，我们有一些小小的建议。

- 尽可能选择静态批处理，但得时刻小心对内存的消耗，并且记住经过静态批处理的物体不可以再被移动。

- 如果无法进行静态批处理，而要使用动态批处理的话，那么请小心上面提到的各种条件限制。例如，尽可能让这样的物体少并且尽可能让这些物体包含少量的顶点属性和顶点数目。
- 对于游戏中的小道具，例如可以捡拾的金币等，可以使用动态批处理。
- 对于包含动画的这类物体，我们无法全部使用静态批处理，但其中如果有不动的部分，可以把这部分标识成“Static”。

除了上述提示外，在使用批处理时还有一些需要注意的地方。由于批处理需要把多个模型变换到世界空间下再合并它们，因此，如果shader中存在一些基于模型空间下的坐标的运算，那么往往会得到错误的结果。一个解决方法是，在shader中使用**DisableBatching**标签来强制使用该Shader的材质不会被批处理。另一个注意事项是，使用半透明材质的物体通常需要使用严格的从后往前的绘制顺序来保证透明混合的正确性。对于这些物体，Unity会首先保证它们的绘制顺序，再尝试对它们进行批处理。这意味着，当绘制顺序无法满足时，批处理无法在这些物体上被成功应用。

尽管在Unity 5.2中，只实现了对一些渲染部分的批处理。而诸如渲染摄像机的深度纹理等部分，还没有实现批处理。但我们相信，在后续的Unity版本中，批处理会应用到越来越多的渲染部分中。

## 16.5 减少需要处理的顶点数目

尽管draw call是一个重要的性能指标，但顶点数目同样有可能成为GPU的性能瓶颈。在本节中，我们将给出3个常用的顶点优化策略。



## 16.5.1 优化几何体

3D游戏制作通常都是由模型制作开始的。而在建模时，有一条规则我们需要记住：尽可能减少模型中三角面片的数目，一些对于模型没有影响、或是肉眼非常难察觉到区别的顶点都要尽可能去掉。为了尽可能减少模型中的顶点数目，美工人员往往需要优化网格结构。在很多三维建模软件中，都有相应的优化选项，可以自动优化网格结构。

在Unity的渲染统计窗口中，我们可以查看到渲染当前帧需要的三角面片数目和顶点数目。需要注意的是，Unity中显示的数目往往要多于建模软件里显示的顶点数，通常Unity中显示的数目要大很多。谁才是对的呢？其实，这是因为在不同的角度上计算的，都有各自的道理，但我们真正应该关心的是Unity里显示的数目。

我们在这里简单解释一下造成这种不同的原因。三维软件更多地是站在我们人类的角度理解顶点的，即组成几何体的每一个点就是一个单独的点。而Unity是站在GPU的角度上去计算顶点数的。在GPU看来，有时需要把一个顶点拆分成两个或更多的顶点。这种将顶点一分为多的原因主要有两个：一个是为了**分离纹理坐标（uv splits）**，另一个是为了**产生平滑的边界（smoothing splits）**。它们的本质，其实都是因为对于GPU来说，顶点的每一个属性和顶点之间必须是一一对应的关系。而分离纹理坐标，是因为建模时一个顶点的纹理坐标有多个。例如，对于一个立方体，它的6个面之间虽然使用了一些相同的顶点，但在不同面上，同一个顶点的纹理坐标可能并不相同。对于GPU来说，这是不可理解的，因此，它必须把这个顶点拆分成多个具有不同纹理坐标的顶点。而平滑边界也是类似的，不同的是，此时一个顶点



可能会对对应多个法线信息或切线信息。这通常是因为我们要决定一个边是一条硬边（hard edge）还是一条平滑边（smooth edge）。

对于GPU来说，它本质上只关心有多少个顶点。因此，尽可能减少顶点的数目其实才是我们真正需要关心的事情。因此，最后一条几何体优化建议就是：移除不必要的硬边以及纹理衔接，避免边界平滑和纹理分离。

### 16.5.2 模型的LOD技术

另一个减少顶点数目的方法是使用LOD（Level of Detail）技术。这种技术的原理是，当一个物体离摄像机很远时，模型上的很多细节是无法被察觉到的。因此，LOD允许当对象逐渐远离摄像机时，减少模型上的面片数量，从而提高性能。

在Unity中，我们可以使用LOD Group组件来为一个物体构建一个LOD。我们需要为同一个对象准备多个包含不同细节程度的模型，然后把它们赋给LOD Group组件中的不同等级，Unity就会自动判断当前位置上需要使用哪个等级的模型。

### 16.5.3 遮挡剔除技术

我们最后要介绍的顶点优化策略就是遮挡剔除（Occlusion culling）技术。遮挡剔除可以用来消除那些在其他物件后面看不到的物件，这意味着资源不会浪费在计算那些看不到的顶点上，进而提升性能。

我们需要把遮挡剔除和摄像机的视锥体剔除（**Frustum Culling**）区分开来。视锥体剔除只会剔除掉那些不在摄像机的视野范围内的对象，但不会判断视野中是否有物体被其他物体挡住。而遮挡剔除会使用一个虚拟的摄像机来遍历场景，从而构建一个潜在可见的对象集合的层级结构。在运行时刻，每个摄像机将会使用这个数据来识别哪些物体是可见的，而哪些被其他物体挡住不可见。使用遮挡剔除技术，不仅可以减少处理的顶点数目，还可以减少**overdraw**，提高游戏性能。

要在Unity中使用遮挡剔除技术，我们需要进行一系列额外的处理工作。具体步骤可以参见Unity手册的相关内容

（<http://docs.unity3d.com/Manual/OcclusionCulling.html>），本书不再赘述。

模型的LOD技术和遮挡剔除技术可以同时减少CPU和GPU的负荷。CPU可以提交更少的**draw call**，而GPU需要处理的顶点和片元数目也减少了。

## 16.6 减少需要处理的片元数目

另一个造成GPU瓶颈的是需要处理过多的片元。这部分优化的重点在于减少**overdraw**。简单来说，**overdraw**指的就是同一个像素被绘制了多次。

Unity还提供了查看**overdraw**的视图，我们可以在Scene视图左上方的下拉菜单中选中**Overdraw**即可。实际上，这里的视图只是提供了查看物体相互遮挡的层数，并不是真正的最终屏幕绘制的**overdraw**。也就是说，可以理解为它显示的是，如果没有使用任何深度测试和其他优

化策略时的`overdraw`。这种视图是通过把所有对象都渲染成一个透明的轮廓，通过查看透明颜色的累计程度，来判断物体之间的遮挡。当然，我们可以使用一些措施来防止这种最坏情况的出现。

### 16.6.1 控制绘制顺序

为了最大限度地避免`overdraw`，一个重要的优化策略就是控制绘制顺序。由于深度测试的存在，如果我们可以保证物体都是从前往后绘制的，那么就可以很大程度上减少`overdraw`。这是因为，在后面绘制的物体由于无法通过深度测试，因此，就不会再进行后面的渲染处理。

在Unity中，那些渲染队列数目小于2 500（如“Background”“Geometry”和“AlphaTest”）的对象都被认为是不透明（`opaque`）的物体，这些物体总体上是从前往后绘制的，而使用其他的队列（如“Transparent”“Overlay”等）的物体，则是从后往前绘制的。这意味着，我们可以尽可能地把物体的队列设置为不透明物体的渲染队列，而尽量避免使用半透明队列。

而且，我们还可以充分利用Unity的渲染队列来控制绘制顺序。例如，在第一人称射击游戏中，对于游戏中的主要人物角色来说，他们使用的`shader`往往比较复杂，但是，由于他们通常会挡住屏幕的很大一部分区域，因此我们可以先绘制它们（使用更小的渲染队列）。而对于一些敌方角色，它们通常会出现在各种掩体后面，因此，我们可以在所有常规的不透明物体后面渲染它们（使用更大的渲染队列）。而对于天空盒子来说，它几乎覆盖了所有的像素，而且我们知道它永远会出现在所有物体的后面，因此，它的队列可以设置为“Geometry+1”。这样，就可以保证不会因为它而造成`overdraw`。

这些排序的思想往往可以节省掉很多渲染时间。

### 16.6.2 时刻警惕透明物体

对于半透明对象来说，由于它们没有开启深度写入，因此，如果要得到正确的渲染效果，就必须从后往前渲染。这意味着，半透明物体几乎一定会造成`overdraw`。如果我们不注意这一点，在一些机器上可能会造成严重的性能下降。例如，对于GUI对象来说，它们大多被设置成了半透明，如果屏幕中GUI占据的比例太多，而主摄像机又没有进行调整而是投影整个屏幕，那么GUI就会造成大量`overdraw`。

因此，如果场景中包含了大面积的半透明对象，或者有很多层相互覆盖的半透明对象（即便它们每个的面积可能都不大），或者是透明的粒子效果，在移动设备上也会造成大量的`overdraw`。这是应该尽量避免的。

对于上述GUI的这种情况，我们可以尽量减少窗口中GUI所占的面积。如果实在无能为力，我们可以把GUI的绘制和三维场景的绘制交给不同的摄像机，而其中负责三维场景的摄像机的视角范围尽量不要和GUI的相互重叠。当然，这样会对游戏的美观度产生一定影响，因此，我们可以在代码中对机器的性能进行判断，例如，首先关闭一些耗费性能的功能，如果发现这个机器表现非常良好，再尝试开启一些特效功能。

在移动平台上，透明度测试也会影响游戏性能。虽然透明度测试没有关闭深度写入，但由于它的实现使用了`discard`或`clip`操作，而这些操作会导致一些硬件的优化策略失效。例如，我们之前讲过PowerVR

使用的基于瓦片的延迟渲染技术，为了减少`overdraw`它会在调用片元着色器前就判断哪些瓦片被真正渲染的。但是，由于透明度测试在片元着色器中使用了`discard`函数改变了片元是否会被渲染的结果，因此，GPU就无法使用上述的优化策略了。也就是说，只有在执行了所有的片元着色器后，GPU才知道哪些片元会被真正渲染到屏幕上，这样，原先那些可以减少`overdraw`的优化就都无效了。这种时候，使用透明度混合的性能往往比使用透明度测试更好。

### 16.6.3 减少实时光照和阴影

实时光照对于移动平台是一种非常昂贵的操作。如果场景中包含了过多的点光源，并且使用了多个Pass的Shader，那么很有可能会造成性能下降。例如，一个场景里如果包含了3个逐像素的点光源，而且使用了逐像素的Shader，那么很有可能将draw call数目（CPU的瓶颈）提高3倍，同时也会增加`overdraw`（GPU的瓶颈）。这是因为，对于逐像素的光源来说，被这些光源照亮的物体需要被再渲染一次。更糟糕的是，无论是静态批处理还是动态批处理，对于这种额外的处理逐像素光源的Pass都无法进行批处理，也就是说，它们会中断批处理。

当然，游戏场景还是需要光照才能得到出色的画面效果。我们看到很多成功的移动平台的游戏，它们的画面效果看起来好像包含了很多光源，但其实这都是骗人的。这些游戏往往使用了烘焙技术，把光照提前烘焙到一张光照纹理（lightmap）中，然后在运行时刻只需要根据纹理采样得到光照结果即可。另一个模拟光源的方法是使用God Ray。场景中很多小型光源的效果都是靠这种方法模拟的。它们一般并不是真的光源，很多情况是通过透明纹理模拟得到的。更多信息可以参见本章的扩展阅读部分。在移动平台上，一个物体使用的逐像素光

源数目应该小于1（不包括平行光）。如果一定要使用更多的实时光，可以选择用逐顶点光照来代替。

在游戏《ShadowGun》中，游戏角色看起来使用了非常复杂高级的光照计算，但这实际上是优化后的结果。开发者们把复杂的光照计算存储到一张查找纹理（lookup texture，也被称为查找表，lookup table，LUT）中。然后在运行时刻，我们只需要使用光源方向、视角方向、法线方向等参数，对LUT采样得到光照结果即可。使用这样的查找纹理，不仅可以让我们使用更出色的光照模型，例如，更加复杂的BRDF模型，还可以利用查找纹理的大小来进一步优化性能，例如，主要角色可以使用更大分辨率的LUT，而一些NPC就使用较小的LUT。《ShadowGun》的开发者开发了一个LUT烘焙工具，来帮助美工人员快速调整光照模型，并把结果存储到LUT中。

实时阴影同样是一个非常消耗性能的效果。不仅是CPU需要提交更多的draw call，GPU也需要进行更多的处理。因此，我们应该尽量减少实时阴影，例如，使用烘焙把静态物体的阴影信息存储到光照纹理中，而只对场景中的动态物体使用适当的实时阴影。

## 16.7 节省带宽

大量使用未经压缩的纹理以及使用过大的分辨率都会造成由于带宽而引发的性能瓶颈。

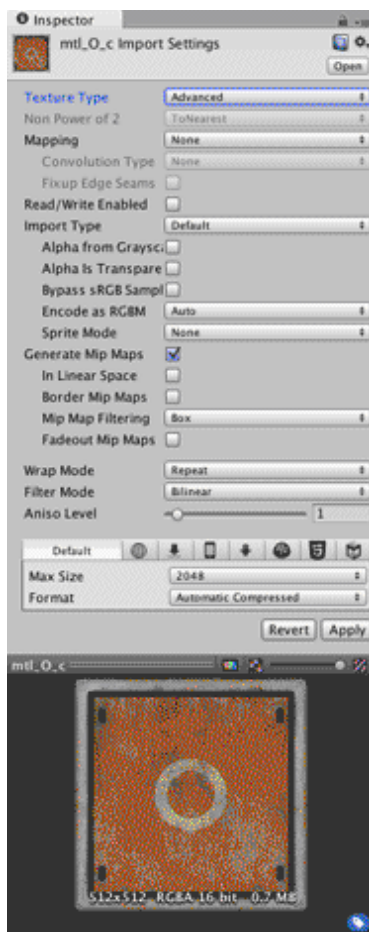
### 16.7.1 减少纹理大小



之前提到过，使用纹理图集可以帮助我们减少draw call的数目，而这些纹理的大小同样是一个需要考虑的问题。需要注意的是，所有纹理的长宽比最好是正方形，而且长宽值最好是2的整数幂。这是因为有很多优化策略只有在这种时候才可以发挥最大效用。在Unity 5中，即便我们导入的纹理长宽值并不是2的整数幂，Unity也会自动把长宽转换到离它最近的2的整数幂值。但我们仍然应该在制作美术资源时就把这条规则谨记在心，防止由于放缩而造成不好的影响。

除此之外，我们还应该尽可能使用多级渐远纹理技术（mipmapping）和纹理压缩。在Unity中，我们可以通过纹理导入面板来查看纹理的各个导入属性。通过把纹理类型设置为**Advanced**，就可以自定义许多选项，例如，是否生成多级渐远纹理（mipmaps），如图16.12所示。当勾选了**Generate Mip Maps**选项后，Unity就会为同一张纹理创建出很多不同大小的小纹理，构成一个纹理金字塔。而在游戏运行中就可以根据距离物体的远近，来动态选择使用哪一个纹理。这是因为，在距离物体很远的时候，就算我们使用了非常精细的纹理，但肉眼也是分辨不出来的。这种时候，我们完全可以使用更小、更模糊的纹理来代替，这可以让GPU使用分辨率更小的纹理，大量节省访问的像素数目。在某些设备上，关闭多级渐远纹理往往会造成严重的性能问题。因此，除非我们确定该纹理不会发生缩放，例如GUI和2D游戏中使用的纹理等，都应该为纹理生成相应的多级渐远纹理。





▲ 图16.12 Unity的高级纹理设置面板

纹理压缩同样可以节省带宽。但对于像Android这样的平台，有很多不同架构的GPU，纹理压缩就变得有点复杂，因为不同的GPU架构有它自己的纹理压缩格式，例如，PowerVRAM的PVRTC格式、Tegra的DXT格式、Adreno的ATC格式。所幸的是，Unity可以根据不同的设备选择不同的压缩格式，而我们只需要把纹理压缩格式设置为自动压缩即可。但是，GUI类型的纹理同样是个例外，一些时候由于对画质的要求，我们不希望对这些纹理进行压缩。

## 16.7.2 利用分辨率缩放

过高的屏幕分辨率也是造成性能下降的原因之一，尤其是对于很多低端手机，除了分辨率高其他硬件条件并不尽如人意，而这恰恰是游戏性能的两个瓶颈：过大的屏幕分辨率和糟糕的GPU。因此，我们可能需要对于特定机器进行分辨率的放缩。当然，这样可能会造成游戏效果的下降，但性能和画面之间永远是个需要权衡的话题。

在Unity中设置屏幕分辨率可以直接调用Screen.SetResolution。实际使用中可能会遇到一些情况，雨松MOMO有一篇文章

（<http://www.xuanyusong.com/archives/3205>）详细讲解了如何使用这种技术，读者可参考。

## 16.8 减少计算复杂度

计算复杂度同样会影响游戏的渲染性能。在本节中，我们会介绍两个方面的技术来减少计算复杂度。

### 16.8.1 Shader的LOD技术

和16.5.2节提到的模型的LOD技术类似，Shader的LOD技术可以控制使用的Shader等级。它的原理是，只有Shader的LOD值小于某个设定的值，这个Shader才会被使用，而使用了那些超过设定值的Shader的物体将不会被渲染。

我们通常会在SubShader中使用类似下面的语句来指明该shader的LOD值：

```
SubShader {  
    Tags { "RenderType"="Opaque" }  
    LOD 200
```

我们也可以在Unity Shader的导入面板上看到该Shader使用的LOD值。在默认情况下，允许的LOD等级是无限大的。这意味着，任何被当前显卡支持的Shader都可以被使用。但是，在某些情况下我们可能需要去掉一些使用了复杂计算的Shader渲染。这时，我们可以使用Shader.maximumLOD或Shader.globalMaximumLOD来设置允许的最大LOD值。

Unity内置的Shader使用了不同的LOD值，例如，Diffuse的LOD为200，而Bumped Specular的LOD为400。

## 16.8.2 代码方面的优化

在实现游戏效果时，我们可以选择在哪里进行某些特定的运算。通常来讲，游戏需要计算的对象、顶点和像素的数目排序是对象数 < 顶点数 < 像素数。因此，我们应该尽可能地把计算放在每个对象或逐顶点上。例如，在第13章实现高斯模糊和边缘检测时，我们把采样坐标的计算放在了顶点着色器中，这样的做法远好于把它们放在片元着色器中。

而在具体的代码编写上，不同的硬件甚至需要不同的处理。因此，一些普遍的规则在某些硬件上可能并不成立。更不幸的是，通常Shader代码的优化并不那么直观，尤其是一些平台上缺少相关的分析器，例如iOS平台。尽管如此，在本节我们还是会给出一些被认为是普遍成立的优化策略，但读者如果发现在某些设备上性能反而有所下降的话，这并不奇怪。

首先第一点是，尽可能使用低精度的浮点值进行运算。最高精度的float/highp适用于存储诸如顶点坐标等变量，但它的计算速度是最慢的，我们应该尽量避免在片元着色器中使用这种精度进行计算。而half/mediump适用于一些标量、纹理坐标等变量，它的计算速度大约是float的两倍。而fixed/lowp适用于绝大多数颜色变量和归一化后的方向矢量，在进行一些对精度要求不高的计算时，我们应该尽量使用这种精度的变量。它的计算速度大约是float的4倍，但要避免对这些低精度变量进行频繁的swizzle操作（如color.xwxw）。还需要注意的是，我们应当尽量避免在不同精度之间的转换，这有可能会造成一定的性能下降。

对于绝大多数GPU来说，在使用插值寄存器把数据从顶点着色器传递给下一个阶段时，我们应该使用尽可能少的插值变量。例如，如果需要对两个纹理坐标进行插值，我们通常会把它们打包在同一个float4类型的变量中，两个纹理坐标分别对应了xy分量和zw分量。然而，对于PowerVR平台来说，这种插值变量是非常廉价的，直接把不同的纹理坐标存储在不同的插值变量中，有时反而性能更好。尤其是，如果在PowerVR上使用类似tex2D(\_MainTex, uv.zw)这样的语句来进行纹理采样，GPU就无法进行一些纹理的预读取，因为它会认为这些纹理采样是需要依赖其他数据的。因此，如果我们特别关心游戏在PowerVR上的性能，就不应该把两个纹理坐标打包在同一个四维变量中。

尽可能不要使用全屏的屏幕后处理效果。如果美术风格实在是需要使用类似Bloom、热扰动这样的屏幕特效，我们应该尽量使用fixed/lowp进行低精度运算（纹理坐标除外，可以使用

half/mediump)。那些高精度的运算可以使用查找表（LUT）或者转移到顶点着色器中进行处理。除此之外，尽量把多个特效合并到一个Shader中。例如，我们可以把颜色校正和添加噪声等屏幕特效在Bloom特效的最后一个Pass中进行合成。还有一个方法就是使用16.8.3节中介绍的缩放思想，来选择性地开启特效。

还有一些读者经常会听到的代码优化规则。

- 尽可能不要使用分支语句和循环语句。
- 尽可能避免使用类似sin、tan、pow、log等较为复杂的数学运算。我们可以使用查找表来作为替代。
- 尽可能不要使用discard操作，因为这会影响硬件的某些优化。

### 16.8.3 根据硬件条件进行缩放

诸如iOS和Android这样的移动平台，不同设备之间的性能千差万别。我们很容易可以找到一台手机的渲染性能是另一台手机的10倍。那么，如何确保游戏可以同时流畅地运行在不同性能的移动设备上呢？一个非常简单且实用的方式是使用所谓的放缩（scaling）思想。我们首先保证游戏最基本的配置可以在所有的平台上运行良好，而对于一些具有更高表现能力的设备，我们可以开启一些更“养眼”的效果，比如使用更高的分辨率，开启屏幕后处理特效，开启粒子效果等。

## 16.9 扩展阅读

Unity官方手册的**移动平台优化实践指南**  
(<http://docs.unity3d.com/Manual/MobileOptimization>)

PracticalGuide.html) 一文给出了一些针对移动平台的优化技术, 包括渲染和图形方面的优化, 以及脚本优化等。手册中另一个针对图像性能优化的文档是**优化图像性能** (<http://docs.unity3d.com/Manual/OptimizingGraphicsPerformance.html>) 一文, 在这个文档中, Unity给出了常见的性能瓶颈以及一些相应的优化技术。除此之外, 文档列出了一个清单, 包含了优化游戏性能的常见做法和约束。

在SIGGRAPH 2011上, Unity进行了一个关于移动平台上Shader优化的演讲 (<http://blogs.unity3d.com/2011/08/18/fast-mobile-shaders-talk-at-siggraph/>)。在这个演讲中, 作者给出了各个主流移动GPU的架构特点, 并给出了相应的shader优化细节, 还结合了真实的Unity游戏项目来进行实例学习。在Unite 2013会议上, Unity呈现了一个名为**针对移动平台优化Unity游戏**的演讲, 在这个简短的演讲中, 作者对造成性能瓶颈的原因进行了分类, 并给出了一些常见的优化技术。在GDC 2014上, Unity展示了如何使用内置的分析器分析移动平台的游戏性能, 读者可以在Youtube上找到相应的视频。在最近的SIGGRAPH 2015会议上, Unity进行了一系列演讲和课程。在Unity和来自高通、ARM等公司的开发人员共同呈现的名为**Moving Mobile Graphics**的课程中, 来自Unity的Renaldas Zioma讲解了移动平台上PBR的优化技术。更多Unity在SIGGRAPH 2015上的演讲, 读者可以参见Unity的博客。

除了手册和演讲资料外, 成功的移动平台中的游戏同样是非常好的学习资料。《ShadowGun》是由MadFinger在2011年发布的一款移动平台的第三人称射击游戏, 使用的开发工具正是Unity。在Unite 2011上, 该游戏的开发者给出了《ShadowGun》中使用的渲染和优化技术, 读者可以在Youtube上面找到这个视频。更难能可贵的是, 在2012

年，《ShadowGun》的开发者放出了示例场景，来让更多的开发者学习如何优化移动平台上的shader。另一个非常好的游戏优化实例是Unity自带的项目《Angry Bots》，读者可以直接在Unity资源商店下载到完整的项目源代码。



## 第5篇 扩展篇

扩展篇旨在进一步扩展读者的视野。本篇将会介绍Unity的表面着色器的实现机制，并介绍基于物理渲染的相关内容。最后，我们给出了更多的关于学习渲染的资料。

### 第17章 Unity的表面着色器探秘

本章将会介绍这些表面着色器是如何实现的，以及我们如何使用这些表面着色器来实现渲染。

### 第18章 基于物理的渲染

这一章将介绍基于物理渲染的理论基础，并解释Unity是如何实现基于物理渲染的。

### 第19章 Unity 5更新了什么

本章将给出Unity 5中一些重要的更新，来帮助读者解决在升级Unity 5时所面对的各种问题。

### 第20章 还有更多内容吗

在最后一章中，我们将给出许多非常有价值的学习资料，来帮助读者进行更深入的学习。

## 第17章 Unity的表面着色器探秘

在2009年的时候（当时Unity的版本是2.x），Unity的渲染工程师Aras（就是经常活跃在论坛和各种会议上的，大名鼎鼎的Aras Pranckevičius）连续发表了3篇名为《Shaders must die》的博客。在这些博客里，Aras认为，把渲染流程分为顶点和像素的抽象层面是错误的，是一种不易理解的抽象。目前，这种在顶点/几何/片元着色器上的操作是对硬件友好的一种方式，但不符合我们人类的思考方式。相反，他认为，应该划分成表面着色器、光照模型和光照着色器这样的层面。其中，表面着色器定义了模型表面的反射率、法线和高光等，光照模型则选择是使用兰伯特还是Blinn-Phong等模型。而光照着色器负责计算光照衰减、阴影等。这样，绝大部分时间我们只需要和表面着色器打交道，例如，混合纹理和颜色等。光照模型可以是提前定义好的，我们只需要选择哪种预定义的光照模型即可。而光照着色器一旦由系统实现后，更不会被轻易改动，从而大大减轻了Shader编写者的工作量。有了这样的想法，Aras在随后的文章中开始尝试把表面着色器整合到Unity中。最终，在2010年的Unity 3中，**Surface Shader**被加入到Unity的大家族中了。

虽然Unity换了一个新的“马甲”，但**表面着色器（Surface Shader）**实际上就是在顶点/片元着色器之上又添加了一层抽象。按Aras的话来解释就是，顶点/几何/片元着色器是硬件能“理解”的渲染方式，而开发者应该使用一种更容易理解的方式。很多时候，使用表面着色器，我们只需要告诉Shader：“嘿，使用这些纹理去填充颜色，使用这个法线

纹理去填充表面法线，使用兰伯特光照模型，其他的就不要来烦我了！”我们不需要考虑是使用前向渲染路径还是延迟渲染路径，场景中有多少光源，它们的类型是什么，怎样处理这些光源，每个Pass需要处理多少个光源等问题（正是因为有这些事情，人们总会抱怨写一个Shader是多么的麻烦.....）。这时，Unity说：“不要急，我来干！”

那么，表面着色器到底长什么样呢？它们又是如何工作的呢？这正是本章要学习的内容。

## 17.1 表面着色器的一个例子

在学习原理之前，我们首先来看一下一个表面着色器长什么样子。为此，我们需要做如下的准备工作。

（1）在Unity中新创建一个场景。在本书资源中，该场景名为Scene\_17\_1。在Unity 5.2中，默认情况下场景将包含一个摄像机和一个平行光，并且使用了内置的天空盒子。在Window → Lighting → Skybox中去掉场景中的天空盒子。

（2）新创建一个材质。在本书资源中，该材质名为BumpedSpecularMat。

（3）新创建一个Unity Shader。在本书资源中，该Unity Shader名为Chapter17-BumpedDiffuse。把新的Unity Shader赋给第2步中创建的材质。

(4) 在场景中创建一个胶囊体 (capsule)，并把第2步中的材质赋给该胶囊体。

(5) 保存场景。

我们将使用表面着色器来实现一个使用了法线纹理的漫反射效果。这可以参考Unity内置的“Legacy Shaders/Bumped Diffuse”的代码实现（可以在官方网站的内置Shader包中找到）。打开Chapter17-BumpedDiffuse，删除原有的代码，把下面的代码粘贴进去：

```
Shader "Unity Shaders Book/Chapter 17/Bumped Diffuse" {
    Properties {
        _Color ("Main Color", Color) = (1,1,1,1)
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _BumpMap ("Normalmap", 2D) = "bump" {}
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 300

        CGPROGRAM
        #pragma surface surf Lambert
        #pragma target 3.0

        sampler2D _MainTex;
        sampler2D _BumpMap;
        fixed4 _Color;

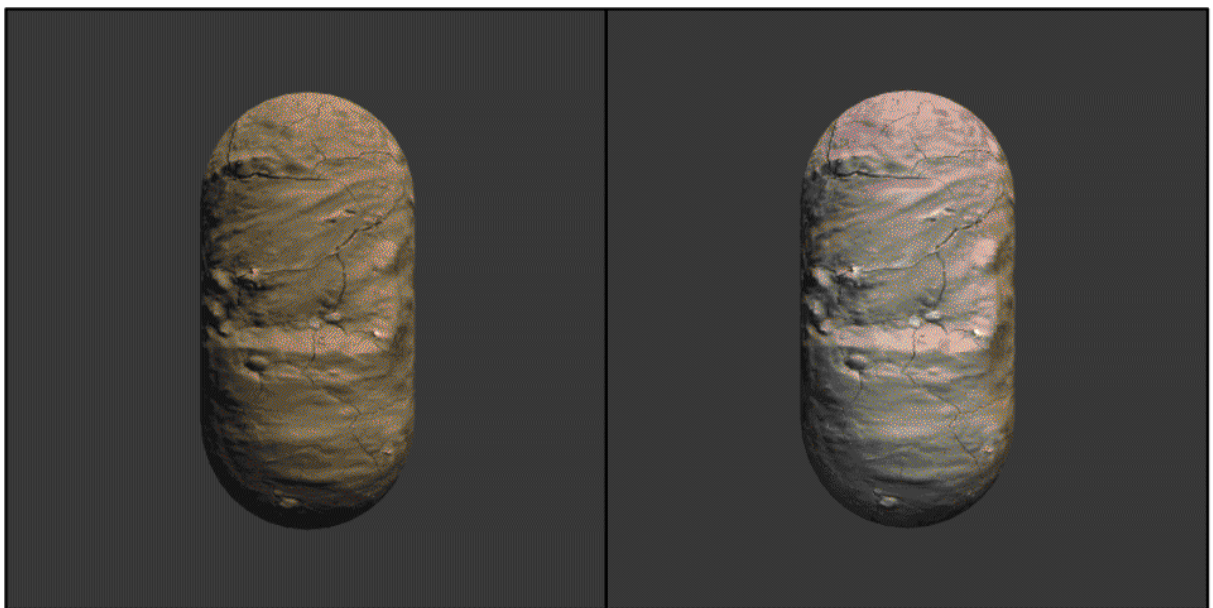
        struct Input {
            float2 uv_MainTex;
            float2 uv_BumpMap;
        };

        void surf (Input IN, inout SurfaceOutput o) {
            fixed4 tex = tex2D(_MainTex, IN.uv_MainTex);
            o.Albedo = tex.rgb * _Color.rgb;
            o.Alpha = tex.a * _Color.a;
            o.Normal = UnpackNormal(tex2D(_BumpMap,
IN.uv_BumpMap));
        }

        ENDCG
    }
}
```

```
} FallBack "Legacy Shaders/Diffuse"
```

保存程序后，返回Unity中查看。在BumpedDiffuseMat的面板上，我们把本书资源中的Assets/Textures/Chapter17/Mud\_Diffuse.tif和Assets/Textures/Chapter17/Mud\_Normal.tif分别拖曳到\_MainTex和\_BumpMap属性上，就可以得到类似图17.1中左图的结果。我们还可以向场景中添加一些点光源和聚光灯，并改变它们的颜色，就可以得到类似图17.1中右图的结果。注意，在这个过程中，我们不需要对代码做任何改动。



▲ 图17.1 表面着色器的例子左边：在一个平行光下的效果。右边：添加了一个点光源（蓝色）和一个聚光灯（紫色）后的效果

从上面的例子可以看出，相比之前所学的顶点/片元着色器技术，表面着色器的代码量很少（只需要三十多行），如果我们使用顶点/片元着色器来实现上述的功能，大概需要150多行代码（参考本书资源中

的“Unity Shaders Book/Common/Bumped Diffuse”)！而且，我们可以非常轻松地实现常见的光照模型，甚至不需要和任何光照变量打交道，Unity就帮我们处理好了每个光源的光照结果。

读者可以在Unity官方手册的**表面着色器的例子**一文 (<http://docs.unity3d.com/Manual/SL-SurfaceShaderExamples.html>) 中找到更多的示例程序。下面，我们将具体学习表面着色器的特点和工作原理。

和顶点/片元着色器需要包含到一个特定的Pass块中不同，表面着色器的CG代码是直接而且也必须写在SubShader块中，Unity会在背后为我们生成多个Pass。当然，可以在SubShader一开始处使用**Tags**来设置该表面着色器使用的标签。在Chapter17-BumpedDiffuse中，我们还使用**LOD**命令设置了该表面着色器的LOD值（详见16.8.1节）。然后，我们使用**CGPROGRAM**和**ENDCG**定义了表面着色器的具体代码。

一个表面着色器中最重要的部分是**两个结构体**以及它的**编译指令**。其中，两个结构体是表面着色器中不同函数之间信息传递的桥梁，而编译指令是我们和Unity沟通的重要手段。

## 17.2 编译指令

我们首先来看一下表面着色器的编译指令。编译指令是我们和Unity沟通的重要方式，通过它可以告诉Unity：“嘿，用这个表面函数设置表面属性，用这个光照模型模拟光照，我不要阴影和环境光，不要雾效！”只需要一句代码，我们就可以完成这么多事情！

编译指令最重要的作用是指明该表面着色器使用的**表面函数**和**光照函数**，并设置一些可选参数。表面着色器的CG块中的第一句代码往往就是它的编译指令。编译指令的一般格式如下：

```
#pragma surface surfaceFunction lightModel [optionalparams]
```

其中，**#pragma surface**用于指明该编译指令是用于定义表面着色器的，在它的后面需要指明使用的表面函数（**surfaceFunction**）和光照模型（**lightModel**），同时，还可以使用一些可选参数来控制表面着色器的一些行为。

### 17.2.1 表面函数

我们之前说过，表面着色器的优点在于抽象出了“表面”这一概念。与之前遇到的顶点/片元抽象层不同，一个对象的表面属性定义了它的反射率、光滑度、透明度等值。而编译指令中的**surfaceFunction**就用于定义这些表面属性。**surfaceFunction**通常就是名为**surf**的函数（函数名可以是任意的），它的函数格式是固定的：

```
void surf (Input IN, inout SurfaceOutput o)  
void surf (Input IN, inout SurfaceOutputStandard o)  
void surf (Input IN, inout SurfaceOutputStandardSpecular o)
```

其中，后两个是Unity 5中由于引入了基于物理的渲染而新添加的两种结构体。**SurfaceOutput**、**SurfaceOutputStandard**和**SurfaceOutputStandardSpecular**都是Unity内置的结构体，它们需要配合不同的光照模型使用，我们会在下一节进行更详细地解释。

在表面函数中，会使用输入结构体**Input IN**来设置各种表面属性，并把这些属性存储在输出结构体**SurfaceOutput**、**SurfaceOutputStandard**



或SurfaceOutputStandardSpecular中，再传递给光照函数计算光照结果。读者可以在Unity手册中的**表面着色器的例子**一文

(<http://docs.unity3d.com/Manual/SL-SurfaceShaderExamples.html>) 中找到更多的示例表面函数。

## 17.2.2 光照函数

除了表面函数，我们还需要指定另一个非常重要的函数——光照函数。光照函数会使用表面函数中设置的各种表面属性，来应用某些光照模型，进而模拟物体表面的光照效果。Unity内置了基于物理的光照模型函数**Standard**和**StandardSpecular**（在UnityPBSLighting.cginc文件中被定义），以及简单的非基于物理的光照模型函数**Lambert**和**BlinnPhong**（在Lighting.cginc文件中被定义）。例如，在Chapter17-BumpedDiffuse中，我们就指定了使用Lambert光照函数。

当然，我们也可以定义自己的光照函数。例如，可以使用下面的函数来定义用于前向渲染中的光照函数：

```
// 用于不依赖视角的光照模型，例如漫反射
half4 Lighting<Name> (SurfaceOutput s, half3 lightDir, half atten);
// 用于依赖视角的光照模型，例如高光反射
half4 Lighting<Name> (SurfaceOutput s, half3 lightDir, half3
viewDir, half atten);
```

读者可以在Unity手册的**表面着色器中的自定义光照模型**一文(<http://docs.unity3d.com/Manual/SL-SurfaceShaderLighting.html>) 中找到更全面的自定义光照模型的介绍。而一些例子可以参见手册中的**表面着色器的光照例子**一文 (<http://docs.unity3d.com/Manual/SL-SurfaceShaderLightingExamples.html>)，这篇文档展示了如何使用表面

着色器来自定义常见的漫反射、高光反射、基于光照纹理等常用的光照模型。

### 17.2.3 其他可选参数

在编译指令的最后，我们还可以设置一些可选参数（**optionalparams**）。这些可选参数包含了很多非常有用的指令类型，例如，开启/设置透明度混合/透明度测试，指明自定义的顶点和颜色修改函数，控制生成的代码等。下面，我们选取了一些比较重要和常用的参数进行更深入地说明。读者可以在Unity官方手册的**编写表面着色器**一文（<http://docs.unity3d.com/Manual/SL-SurfaceShaders.html>）中找到更加详细的参数和设置说明。

- 自定义的修改函数。除了表面函数和光照模型外，表面着色器还可以支持其他两种自定义的函数：**顶点修改函数**（**vertex:VertexFunction**）和**最后的颜色修改函数**（**finalcolor:ColorFunction**）。顶点修改函数允许我们自定义一些顶点属性，例如，把顶点颜色传递给表面函数，或是修改顶点位置，实现某些顶点动画等。最后的颜色修改函数则可以在颜色绘制到屏幕前，最后一次修改颜色值，例如实现自定义的雾效等。
- 阴影。我们可以通过一些指令来控制与阴影相关的代码。例如，**addshadow**参数会为表面着色器生成一个阴影投射的Pass。通常情况下，Unity可以直接在**FallBack**中找到通用的光照模式为**ShadowCaster**的Pass，从而将物体正确地渲染到深度和阴影纹理中（详见9.4节）。但对于一些进行了顶点动画、透明度测试的物体，我们就需要对阴影的投射进行特殊处理，来为它们产生正确的阴影，正如我们在11.3.3节中看到的一样。**fullforwardshadows**

参数则可以在前向渲染路径中支持所有光源类型的阴影。默认情况下，Unity只支持最重要的平行光的阴影效果。如果我们需要让点光源或聚光灯在前向渲染中也可以有阴影，就可以添加这个参数。相反地，如果我们不想对使用这个Shader的物体进行任何阴影计算，就可以使用**noshadow**参数来禁用阴影。

- 透明度混合和透明度测试。我们可以通过**alpha**和**alphatest**指令来控制透明度混合和透明度测试。例如，**alphatest:VariableName**指令会使用名为VariableName的变量来剔除不满足条件的片元。此时，我们可能还需要使用上面提到的**addshadow**参数来生成正确的阴影投射的Pass。
- 光照。一些指令可以控制光照对物体的影响，例如，**noambient**参数会告诉Unity不要应用任何环境光照或光照探针（light probe）。**novertexlights**参数告诉Unity不要应用任何逐顶点光照。**noforwardadd**会去掉所有前向渲染中的额外的Pass。也就是说，这个Shader只会支持一个逐像素的平行光，而其他的光源会按照逐顶点或SH的方法来计算光照影响。这个参数通常会用于移动平台版本的表面着色器中。还有一些用于控制光照烘焙、雾效模拟的参数，如**nolightmap**、**nofog**等。
- 控制代码的生成。一些指令还可以控制由表面着色器自动生成的代码，默认情况下，Unity会为一个表面着色器生成相应的前向渲染路径、延迟渲染路径使用的Pass，这会导致生成的Shader文件比较大。如果我们确定该表面着色器只会在某些渲染路径中使用，就可以**exclude\_path:deferred**、**exclude\_path:forward**和**exclude\_path:prepass**来告诉Unity不需要为某些渲染路径生成代码。

从上述可以看出，表面着色器支持的编译指令参数很多，为我们编写表面着色器提供了很大的方便。之前在顶点/片元着色器中需要耗费大量代码来完成的工作，在表面着色器中可能只需要一个参数就可以了。当然，相比与顶点/片元着色器，表面着色器也有它自身的限制，我们会在17.6节中对比它们的优缺点。

## 17.3 两个结构体

在上一节我们已经讲过，表面着色器支持最多自定义4种关键的函数：表面函数（用于设置各种表面性质，如反射率、法线等），光照函数（定义表面使用的光照模型），顶点修改函数（修改或传递顶点属性），最后的颜色修改函数（对最后的颜色进行修改）。那么，这些函数之间的信息传递是怎么实现的呢？例如，我们想把顶点颜色传递给表面函数，添加到表面反射率的计算中，要怎么做呢？这就是两个结构体的工作。

一个表面着色器需要使用两个结构体：表面函数的输入结构体 **Input**，以及存储了表面属性的结构体 **SurfaceOutput**（Unity 5新引入了另外两个同种的结构体 **SurfaceOutputStandard** 和 **SurfaceOutputStandardSpecular**）。

### 17.3.1 数据来源：Input结构体

**Input**结构体包含了许多表面属性的数据来源，因此，它会作为表面函数的输入结构体（如果自定义了顶点修改函数，它还会是顶点修改函数的输出结构体）。**Input**支持很多内置的变量名，通过这些变量名，我们告诉Unity需要使用的数据信息。例如，在Chapter17-

BumpedDiffuse中，Input结构体中包含了主纹理和法线纹理的采样坐标uv\_MainTex和uv\_BumpMap。这些采样坐标必须以“uv”为前缀（实际上也可用“uv2”为前缀，表明使用次纹理坐标集合），后面紧跟纹理名称。以主纹理\_MainTex为例，如果需要使用它的采样坐标，就需要在Input结构体中声明float2 uv\_MainTex来对应它的采样坐标。表17.1列出了Input结构体中内置的其他变量。

表17.1

变 量	描 述
float3 viewDir	包含了视角方向，可用于计算边缘光照等
使用COLOR语义定义的float4变量	包含了插值后的逐顶点颜色
float4 screenPos	包含了屏幕空间的坐标，可以用于反射或屏幕特效
float3 worldPos	包含了世界空间下的位置
float3 worldRefl	包含了世界空间下的反射方向。前提是没有修改表面法线o.Normal
float3 worldRefl; INTERNAL_DATA	如果修改了表面法线o.Normal，需要使用该变量告诉Unity要基于修改后的法线计算世界空间下的反射方向。在表面函数中，我们需要使用WorldReflectionVector(IN, o.Normal)来得到世界空间下的反射方向

变 量	描 述
float3 worldNormal	包含了世界空间的法线方向。前提是没有修改表面法线 o.Normal
float3 worldNormal; INTERNAL_DATA	如果修改了表面法线o.Normal，需要使用该变量告诉Unity要基于修改后的法线计算世界空间下的法线方向。在表面函数中，我们需要使用WorldNormalVector(IN, o.Normal)来得到世界空间下的法线方向

需要注意的是，我们并不需要自己计算上述的各个变量，而只需要在Input结构体中按上述名称严格声明这些变量即可，Unity会在背后为我们准备好这些数据，而我们只需要在表面函数中直接使用它们即可。一个例外情况是，我们自定义了顶点修改函数，并需要向表面函数中传递一些自定义的数据。例如，为了自定义雾效，我们可能需要在顶点修改函数中根据顶点在视角空间下的位置信息计算雾效混合系数，这样我们就可以在Input结构体中定义一个名为half fog的变量，把计算结果存储在该变量后进行输出。

### 17.3.2 表面属性：SurfaceOutput结构体

有了Input结构体来提供所需要的数据后，我们就可以据此计算各种表面属性。因此，另一个结构体就是用于存储这些表面属性的结构体，即**SurfaceOutput**、**SurfaceOutputStandard**和**SurfaceOutputStandardSpecular**，它会作为表面函数的输出，随后会作为光照函数的输入来进行各种光照计算。相比与Input结构体的自由性，这个结构体里面的变量是提前就声明好的，不可以增加也不会减

少（如果没有对某些变量赋值，就会使用默认值）。SurfaceOutput的声明可以在Lighting.cginc文件中找到：

```
struct SurfaceOutput {
    fixed3 Albedo;
    fixed3 Normal;
    fixed3 Emission;
    half Specular;
    fixed Gloss;
    fixed Alpha;
};
```

而SurfaceOutputStandard和SurfaceOutputStandardSpecular的声明可以在UnityPBSLighting.cginc中找到：

```
struct SurfaceOutputStandard
{
    fixed3 Albedo;           // base (diffuse or specular) color
    fixed3 Normal;           // tangent space normal, if written
    half3 Emission;
    half Metallic;           // 0=non-metal, 1=metal
    half Smoothness;         // 0=rough, 1=smooth
    half Occlusion;           // occlusion (default 1)
    fixed Alpha;             // alpha for transparencies
};

struct SurfaceOutputStandardSpecular
{
    fixed3 Albedo;           // diffuse color
    fixed3 Specular;         // specular color
    fixed3 Normal;           // tangent space normal, if written
    half3 Emission;
    half Smoothness;         // 0=rough, 1=smooth
    half Occlusion;           // occlusion (default 1)
    fixed Alpha;             // alpha for transparencies
};
```

在一个表面着色器中，只需要选择上述三者中的其一即可，这取决于我们选择使用的光照模型。Unity内置的光照模型有两种，一种是Unity 5之前的、简单的、非基于物理的光照模型，包括了**Lambert**和**BlinnPhong**；另一种是Unity 5添加的、基于物理的光照模型，包括



**Standard**和**StandardSpecular**，这种模型会更加符合物理规律，但计算也会复杂很多。如果使用了非基于物理的光照模型，我们通常会使用**SurfaceOutput**结构体，而如果使用了基于物理的光照模型**Standard**或**StandardSpecular**，我们会分别使用**SurfaceOutputStandard**或**SurfaceOutput StandardSpecular**结构体。其中，**SurfaceOutputStandard**结构体用于默认的金属工作流程（**Metallic Workflow**），对应了**Standard**光照函数；而**SurfaceOutputStandardSpecular**结构体用于高光工作流程（**Specular Workflow**），对应了**StandardSpecular**光照函数。更多关于基于物理的渲染内容，我们会在第18章中讲到。

在本节，我们着重介绍一下**SurfaceOutput**结构体中的变量和含义。在表面函数中，我们需要根据**Input**结构体传递的各个变量计算表面属性。在**SurfaceOutput**结构体，这些表面属性包括了。

- **fixed3 Albedo**: 对光源的反射率。通常由纹理采样和颜色属性的乘积计算而得。
- **fixed3 Normal**: 表面法线方向。
- **fixed3 Emission**: 自发光。Unity通常会在片元着色器最后输出前（并在最后的顶点函数被调用前，如果定义了的话），使用类似下面的语句进行简单的颜色叠加：

```
c.rgb += o.Emission;
```

- **half Specular**: 高光反射中的指数部分的系数，影响高光反射的计算。例如，如果使用了内置的**BlinnPhong**光照函数，它会使用如下语句计算高光反射的强度：

```
float spec = pow (nh, s.Specular*128.0) * s.Gloss;
```

- **fixed Gloss:** 高光反射中的强度系数。和上面的**Specular**类似，计算公式见上面的代码。一般在包含了高光反射的光照模型里使用。
- **fixed Alpha:** 透明通道。如果开启了透明度的话，会使用该值进行颜色混合。

尽管表面着色器极大地减少了我们的工作量，但它带来的一个问题是，我们经常不知道为什么会得到这样的渲染结果。如果你不是一个“好奇宝宝”的话，你可以高高兴兴地使用表面着色器来方便地实现一些不错的渲染效果。但是，一些好奇的初学者往往会提出这样的问题：“为什么我的场景里没有灯光，但物体不是全黑的呢？为什么我把光源的颜色调成黑色，物体还是有一些渲染颜色呢？”这些问题都源于表面着色器对我们隐藏了实现细节。而想要更加得心应手地使用表面着色器，我们就需要学习它的工作流水线，并了解Unity是如何为一个表面着色器生成对应的顶点/片元着色器的（时刻记着，表面着色器本质上就是包含了很多**Pass**的顶点/片元着色器）。

## 17.4 Unity背后做了什么

在前面的内容中，我们已经了解到如何利用编译指令、自定义函数（表面函数、光照函数，以及可选的顶点修改函数和最后的颜色修改函数）和两个结构体来实现一个表面着色器。我们一直强调，Unity实际会在背后为表面着色器生成真正的顶点/片元着色器。那么，表面着色器中的各个函数、编译指令和结构体与顶点/片元着色器之间有什么关系呢？这正是本节要学习的内容。

我们之前说过，Unity在背后会根据表面着色器生成一个包含了很多Pass的顶点/片元着色器。这些Pass有些是为了针对不同的渲染路径，例如，默认情况下Unity会为前向渲染路径生成**LightMode**为**ForwardBase**和**ForwardAdd**的Pass，为Unity 5之前的延迟渲染路径生成**LightMode**为**PrePassBase**和**PrePassFinal**的Pass，为Unity 5之后的延迟渲染路径生成**LightMode**为**Deferred**的Pass。还有一些Pass是用于产生额外的信息，例如，为了给光照映射和动态全局光照提取表面信息，Unity会生成一个**LightMode**为**Meta**的Pass。有些表面着色器由于修改了顶点位置，因此，我们可以利用**addshadow**编译指令为它生成相应的**LightMode**为**ShadowCaster**的阴影投射Pass。这些Pass的生成都是基于我们在表面着色器中的编译指令和自定义的函数，这是有规律可循的。Unity提供了一个功能，让那些“好奇宝宝”可以对表面着色器自动生成的代码一探究竟：在每个编译完成的表面着色器的面板上，都有一个“Show generated code”的按钮，如图17.2所示，我们只需要单击一下它就可以看到Unity为这个表面着色器生成的所有顶点/片元着色器。

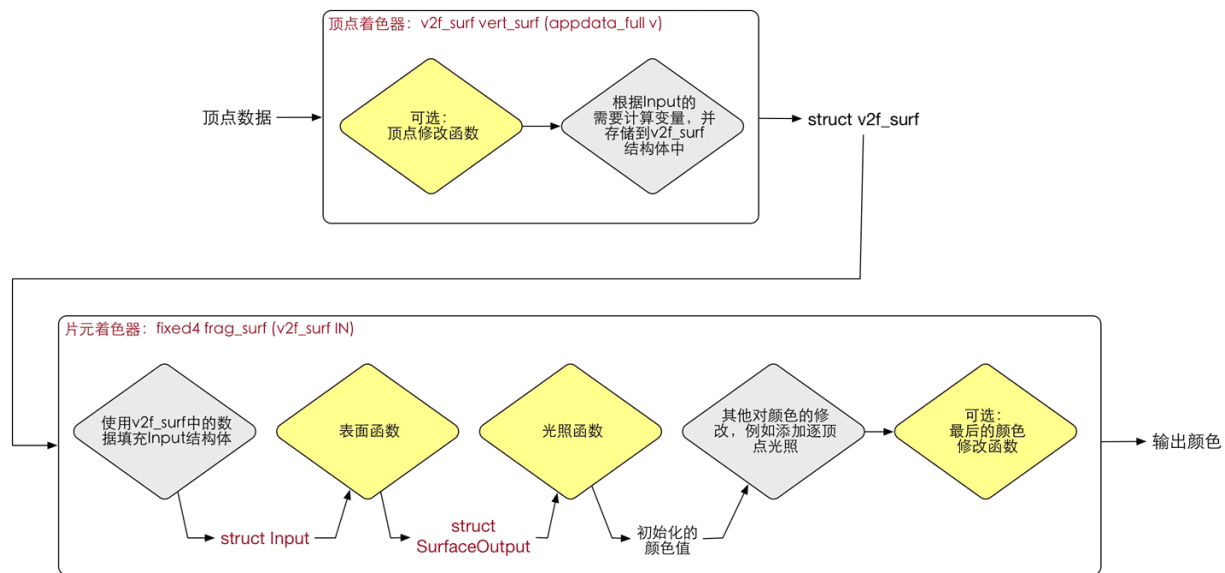


▲ 图17.2 查看表面着色器生成的代码

通过查看这些代码，我们就可以了解到Unity到底是如何根据表面着色器生成各个Pass的。以Unity生成的LightMode为ForwardBase的Pass（用于前向渲染）为例，它的渲染计算流水线如图17.3所示。从图17.3中我们可以看出，4个允许自定义的函数在流水线中的位置。

Unity对该Pass的自动生成过程大致如下。

（1）直接将表面着色器中CGPROGRAM和ENDCG之间的代码复制过来，这些代码包括我们对Input结构体、表面函数、光照函数（如果自定了的话）等变量和函数的定义。这些函数和变量会在之后的处理过程中被当成正常的结构体和函数进行调用。



▲ 图17.3 表面着色器的渲染计算流水线。黄色：可以自定义的函数。灰色：Unity自动生成的计算步骤

（2）Unity会分析上述代码，并据此生成顶点着色器的输出——v2f\_surf结构体，用于在顶点着色器和片元着色器之间进行数据传递。Unity会分析我们在自定义函数中所使用的变量，例如，纹理坐标、视

角方向、反射方向等。如果需要，它就会在v2f\_surf中生成相应的变量。而且，即便有时我们在Input中定义了某些变量（如某些纹理坐标），但Unity在分析后续代码时发现我们并没有使用这些变量，那么这些变量实际上是不会在v2f\_surf中生成的。这也就是说，Unity做了一些优化。v2f\_surf中还包含了一些其他需要的变量，例如阴影纹理坐标、光照纹理坐标、逐顶点光照等。

（3）接着，生成顶点着色器。

① 如果我们自定义了**顶点修改函数**，Unity会首先调用顶点修改函数来修改顶点数据，或填充自定义的Input结构体中的变量。然后，Unity会分析顶点修改函数中修改的数据，在需要时通过Input结构体将修改结果存储到v2f\_surf相应的变量中。

② 计算v2f\_surf中其他生成的变量值。这主要包括了顶点位置、纹理坐标、法线方向、逐顶点光照、光照纹理的采样坐标等。当然，我们可以通过编译指令来控制某些变量是否需要计算。

③ 最后，将v2f\_surf传递给接下来的片元着色器。

（4）生成片元着色器。

① 使用v2f\_surf中的对应变量填充Input结构体，例如，纹理坐标、视角方向等。

② 调用我们自定义的**表面函数**填充SurfaceOutput结构体。

③ 调用**光照函数**得到初始的颜色值。如果使用的是内置的Lambert或BlinnPhong光照函数，Unity还会计算动态全局光照，并添加到光照模型的计算中。

④ 进行其他的颜色叠加。例如，如果没有使用光照烘焙，还会添加逐顶点光照的影响。

⑤ 最后，如果自定义了**最后的颜色修改函数**，Unity就会调用它进行最后的颜色修改。

其他Pass的生成过程和上面类似，在此不再赘述。

## 17.5 表面着色器实例分析

为了帮助读者更加深入地理解表面着色器背后的原理，我们在本节以一个表面着色器为例，分析Unity为它生成的代码。

读者可以在本书资源中的Scene\_17\_4中找到相应的测试场景。它实现的效果是对模型进行膨胀，如图17.4所示。



▲图17.4 沿顶点法线对模型进行膨胀 左边：膨胀前，右边：膨胀后

这种效果的实现非常简单，就是在顶点修改函数中沿着顶点法线方向扩张顶点位置。为了分析表面着色器中4个可自定义函数（顶点修改函数、表面函数、光照函数和最后的颜色修改函数）的原理，在本例中我们对这4个函数全部采用了自定义的实现。读者可以在Chapter17-NormalExtrusion文件中找到该表面着色器，它的代码如下：

```
Shader "Unity Shaders Book/Chapter 17/Normal Extrusion" {
    Properties {
        _ColorTint ("Color Tint", Color) = (1,1,1,1)
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _BumpMap ("Normalmap", 2D) = "bump" {}
        _Amount ("Extrusion Amount", Range(-0.5, 0.5)) = 0.1
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 300

        CGPROGRAM

        // surf - which surface function.
        // CustomLambert - which lighting model to use.
        // vertex:myvert - use custom vertex modification function.
        // finalcolor:mycolor - use custom final color modification
function.
        // addshadow - generate a shadow caster pass. Because we
modify the vertex position,
        // the shader needs special shadows handling.
        // exclude_path:deferred/exclude_path:prepass - do not
generate passes for
        //deferred/legacy deferred rendering path.
        // nometa - do not generate a "meta" pass (that's used by
lightmapping & dynamic
        //global illumination to extract surface information).
        #pragma surface surf CustomLambert vertex:myvert
finalcolor:mycolor addshadow
        exclude_path:deferred exclude_path:prepass nometa
        #pragma target 3.0

        fixed4 _ColorTint;
        sampler2D _MainTex;
        sampler2D _BumpMap;
        half _Amount;
```



```

    struct Input {
        float2 uv_MainTex;
        float2 uv_BumpMap;
    };

    void myvert (inout appdata_full v) {
        v.vertex.xyz += v.normal * _Amount;
    }

    void surf (Input IN, inout SurfaceOutput o) {
        fixed4 tex = tex2D(_MainTex, IN.uv_MainTex);
        o.Albedo = tex.rgb;
        o.Alpha = tex.a;
        o.Normal = UnpackNormal(tex2D(_BumpMap,
IN.uv_BumpMap));
    }

    half4 LightingCustomLambert (SurfaceOutput s, half3
lightDir, half atten) {
        half NdotL = dot(s.Normal, lightDir);
        half4 c;
        c.rgb = s.Albedo * _LightColor0.rgb * (NdotL * atten);
        c.a = s.Alpha;
        return c;
    }

    void mycolor (Input IN, SurfaceOutput o, inout fixed4
color) {
        color *= _ColorTint;
    }

    ENDCG
}

Fallback "Legacy Shaders/Diffuse"
}

```

在顶点修改函数中，我们使用顶点法线对顶点位置进行膨胀；表面函数使用主纹理设置了表面属性中的反射率，并使用法线纹理设置了表面法线方向；光照函数实现了简单的兰伯特漫反射光照模型；在最后的颜色修改函数中，我们简单地使用了颜色参数对输出颜色进行调整。注意，除了4个函数外，我们在`#pragma surface`的编译指令一行中还指定了一些额外的参数。由于我们修改了顶点位置，因此，要对

其他物体产生正确的阴影效果并不能直接依赖FallBack中找到的阴影投射Pass，**addshadow**参数可以告诉Unity要生成一个该表面着色器对应的阴影投射Pass。默认情况下，Unity会为所有支持的渲染路径生成相应的Pass，为了缩小自动生成的代码量，我们使用**exclude\_path:deferred**和**exclude\_path:prepass**来告诉Unity不要为延迟渲染路径生成相应的Pass。最后，我们使用**nometa**参数取消对提取元数据的Pass的生成。

当在该表面着色器的导入面板中单击“Show generated code”按钮后，我们就可以看到Unity生成的顶点/片元着色器了。由于代码比较多，为了节省篇幅我们不再把全部代码粘贴到这里。因此，在往下阅读之前，请读者先打开生成的代码文件，以便明白我们接下来的分析。

在这个将近600行代码的文件中，Unity一共为该表面着色器生成了3个Pass，它们的LightMode分别是ForwardBase、ForwardAdd和ShadowCaster，分别对应了前向渲染路径中的处理逐像素平行光的Pass、处理其他逐像素光的Pass、处理阴影投射的Pass。这些Pass的原理可以回顾9.1.1节和9.4节中的相关内容。读者可以在这些代码中看到大量的**#ifdef**和**#if**语句，这些语句可以判断一些渲染条件，例如，是否使用了动态光照纹理、是否使用了逐顶点光照、是否使用了屏幕空间的阴影等，Unity会根据这些条件来进行不同的光照计算，这正是表面着色器的魅力之一——把这些烦人的光照计算交给Unity来做！

需要注意的是，不同的Unity版本可能生成的代码有少许不同。在本书中，我们以Unity 5.2.1中的结果为准。下面，我们来分析Unity生成

的ForwardBase Pass。

(1) Unity首先指明了一些编译指令:

```
// ---- forward rendering base pass:
Pass {
    Name "FORWARD"
    Tags { "LightMode" = "ForwardBase" }

    CGPROGRAM
    // compile directives
    #pragma vertex vert_surf
    #pragma fragment frag_surf
    #pragma target 3.0
    #pragma multi_compile_fwdbase
    #include "HLSLSupport.cginc"
    #include "UnityShaderVariables.cginc"
```

顶点着色器`vert_surf`和片元着色器`frag_surf`都是自动生成的。

(2) 之后出现的是一些自动生成的注释:

```
    // Surface shader code generated based on:
// vertex modifier: 'myvert'
// writes to per-pixel normal: YES
// writes to emission: no
// needs world space reflection vector: no
// needs world space normal vector: no
// needs screen space position: no
// needs world space position: no
// needs view direction: no
// needs world space view direction: no
// needs world space position for lighting: no
// needs world space view direction for lighting: no
// needs world space view direction for lightmaps: no
// needs vertex color: no
// needs VFACE: no
// passes tangent-to-world matrix to pixel shader: YES
// reads from normal: no
// 2 texcoords actually used
//   float2 _MainTex
//   float2 _BumpMap
```

尽管这些对渲染结果没有影响，但我们可以从这些注释中理解到Unity的分析过程和它的分析结果。

(3) 随后，Unity定义了一些宏来辅助计算：

```
#define INTERNAL_DATA half3 internalSurfaceTtoW0; half3  
internalSurfaceTtoW1; half3 internalSurfaceTtoW2;  
#define WorldReflectionVector(data,normal) reflect (data.worldRefl,  
half3(dot(data.internalSurfaceTtoW0,normal),  
dot(data.internalSurfaceTtoW1,normal), dot(data.  
internalSurfaceTtoW2,normal)))  
#define WorldNormalVector(data,normal)  
fixed3(dot(data.internalSurfaceTtoW0,normal),  
dot(data.internalSurfaceTtoW1,normal),  
dot(data.internalSurfaceTtoW2,normal))
```

实际上，在本例中上述宏并没有被用到。这些宏是为了在修改了表面法线的情况下，辅助计算得到世界空间下的反射方向和法线方向，与之对应的是Input结构体中的一些变量（可参见17.3.1节）。

(4) 接着，Unity把我们在表面着色器中编写的CG代码复制过来，作为Pass的一部分，以便后续调用。

(5) 然后，Unity定义了顶点着色器到片元着色器的插值结构体（即顶点着色器的输出结构体）v2f\_surf。在定义之前，Unity使用#ifdef语句来判断是否使用了光照纹理，并为不同的情况生成不同的结构体。主要的区别是，如果没有使用光照纹理，就需要定义一个存储逐顶点和SH光照的变量。

```
// vertex-to-fragment interpolation data  
// no lightmaps:  
#ifdef LIGHTMAP_OFF  
struct v2f_surf {  
    float4 pos : SV_POSITION;  
    float4 pack0 : TEXCOORD0; // _MainTex _BumpMap  
    fixed3 tSpace0 : TEXCOORD1;
```

```

    fixed3 tSpace1 : TEXCOORD2;
    fixed3 tSpace2 : TEXCOORD3;
    fixed3 vlight : TEXCOORD4; // ambient/SH/vertexlights
    SHADOW_COORDS(5)
    #if SHADER_TARGET >= 30
    float4 lmap : TEXCOORD6;
    #endif
};
#endif
// with lightmaps:
#ifndef LIGHTMAP_OFF
struct v2f_surf {
    float4 pos : SV_POSITION;
    float4 pack0 : TEXCOORD0; // _MainTex _BumpMap
    fixed3 tSpace0 : TEXCOORD1;
    fixed3 tSpace1 : TEXCOORD2;
    fixed3 tSpace2 : TEXCOORD3;
    float4 lmap : TEXCOORD4;
    SHADOW_COORDS(5)
};
#endif

```

上面很多变量名看起来很陌生，但实际上大部分变量的含义我们在之前都碰到过，只是这里使用了不同的名称而已。例如，在下面我们会看到，`pack0`中实际上存储的就是主纹理和法线纹理的采样坐标，而`tSpace0`、`tSpace1`和`tSpace2`存储了从切线空间到世界空间的变换矩阵。一个比较陌生的变量是`vlight`，Unity会把逐顶点和SH光照的结果存储到该变量里，并在片元着色器中和原光照结果进行叠加（如果需要的话）。

(6) 随后，Unity定义了真正的顶点着色器。顶点着色器首先会调用我们自定义的顶点修改函数来修改一些顶点属性：

```

// vertex shader
v2f_surf vert_surf (appdata_full v) {
    v2f_surf o;
    UNITY_INITIALIZE_OUTPUT(v2f_surf,o);
    myvert (v);
}

```

在我们的实现中，只对顶点坐标进行了修改，而不需要向Input结构体中添加并存储新的变量。也可以使用另一个版本的函数声明来把顶点修改函数中的某些计算结果存储到Input结构体中：

```
void vert(inout appdata_full v, out Input o);
```

之后的代码是用于计算v2f\_surf中各个变量的值。例如，计算经过MVP矩阵变换后的顶点坐标；使用TRANSFORM\_TEX内置宏计算两个纹理的采样坐标，并分别存储在o.pack0的xy分量和zw分量中；计算从切线空间到世界空间的变换矩阵，并把矩阵的每一行分别存储在o.tSpace0、o.tSpace1和o.tSpace2变量中；判断是否使用了光照映射和动态光照映射，并在需要时把两种光照纹理的采样坐标计算结果存储在o.lmap.xy和o.lmap.zw分量中；判断是否使用了光照映射，如果没有的话就计算该顶点的SH光照（一种快速计算光照的方法），把结果存储到o.vlight中；判断是否开启了逐顶点光照，如果是就计算最重要的4个逐顶点光照的光照结果，把结果叠加到o.vlight中。这部分代码读者可以在生成的文件中找到，这里不再粘贴出来。

最后，计算阴影坐标并传递给片元着色器：

```
TRANSFER_SHADOW(o); // pass shadow coordinates to pixel shader
return o;
}
```

(7) 在Pass的最后，Unity定义了真正的片元着色器。Unity首先利用插值后的结构体v2f\_surf来初始化Input结构体中的变量：

```
// fragment shader
fixed4 frag_surf (v2f_surf IN) : SV_Target {
    // prepare and unpack data
    Input surfIN;
```

```
UNITY_INITIALIZE_OUTPUT(Input, surfIN);  
surfIN.uv_MainTex = IN.pack0.xy;  
surfIN.uv_BumpMap = IN.pack0.zw;
```

随后，Unity声明了一个SurfaceOutput结构体的变量，并对其中的表面属性进行了初始化，再调用了表面函数：

```
#ifdef UNITY_COMPILER_HLSL  
    SurfaceOutput o = (SurfaceOutput)0;  
#else  
    SurfaceOutput o;  
#endif  
o.Albedo = 0.0;  
o.Emission = 0.0;  
o.Specular = 0.0;  
o.Alpha = 0.0;  
o.Gloss = 0.0;  
  
// call surface function  
surf (surfIN, o);
```

在上面的代码中，Unity还使用#ifdef语句判断当前的编译语言类型是否是HLSL，如果是就使用更严格的声明方式来声明SurfaceOutput结构体（因为DirectX平台往往有更加严格的语义要求）。当对各个表面属性进行初始化后，Unity调用了表面函数surf来填充这些表面属性。

之后，Unity进行了真正的光照计算。首先，计算得到了光照衰减和世界空间下的法线方向：

```
// compute lighting & shadowing factor  
UNITY_LIGHT_ATTENUATION(atten, IN, worldPos)  
fixed4 c = 0;  
fixed3 worldN;  
worldN.x = dot(IN.tSpace0.xyz, o.Normal);  
worldN.y = dot(IN.tSpace1.xyz, o.Normal);  
worldN.z = dot(IN.tSpace2.xyz, o.Normal);  
o.Normal = worldN;
```



其中，变量`c`用于存储最终的输出颜色，此时被初始化为0。随后，Unity判断是否关闭了光照映射，如果关闭了，就把逐顶点的光照结果叠加到输出颜色中：

```
#ifdef LIGHTMAP_OFF
    c.rgb += o.Albedo * IN.vlight;
#endif // LIGHTMAP_OFF
```

而如果需要使用光照映射，Unity就会使用之前计算的光照纹理采样坐标，对光照纹理进行采样并解码，得到光照纹理中的光照结果。这部分代码读者可以在生成的代码中找到，这里不再粘贴过来。

如果没有使用光照映射，意味着我们需要使用自定义的光照模型计算光照结果：

```
// realtime lighting: call lighting function
#ifdef LIGHTMAP_OFF
    c += LightingCustomLambert (o, lightDir, atten);
#else
    c.a = o.Alpha;
#endif
```

而如果使用了光照映射的话，Unity会根据之前由光照纹理得到的结果得到颜色值，并叠加到输出颜色`c`中。如果还开启了动态光照映射，Unity还会计算对动态光照纹理的采样结果，同样把结果叠加到输出颜色`c`中。这部分代码读者可以在生成的代码中找到，这里不再粘贴过来。

最后，Unity调用自定义的颜色修改函数，对输出颜色`c`进行最后的修改：

```
mycolor (surfIN, o, c);
UNITY_OPAQUE_ALPHA(c.a);
```

```
    return c;  
}
```

在上面的代码中，Unity还使用了内置宏UNITY\_OPAQUE\_ALPHA（在UnityCG.cginc里被定义）来重置片元的透明通道。在默认情况下，所有不透明类型的表面着色器的透明通道都会被重置为1.0，而不管我们是否在光照函数中改变了它，如上所示。如果我们想要保留它的透明通道的话，可以在表面着色器的编译指令中添加**keepalpha**参数。

至此，ForwardBase Pass就结束了。接下来的ForwardAdd Pass和上面的ForwardBase Pass基本类似，只是代码更加简单了，Unity去掉了对逐顶点光照和各种判断是否使用了光照映射的代码，因为这些额外的Pass不需要考虑这些。

最后一个重要的Pass是ShadowCaster Pass。相比于之前的两个Pass，它的代码比较简单短小。它的生成原理很简单，就是通过调用自定义的顶点修改函数来保证计算阴影时使用的是和之前一致的顶点坐标。正如我们在11.3.3节和15.1节中看到的一样，这个自定义的阴影投射的Pass同样使用了内置的V2F\_SHADOW\_CASTER、TRANSFER\_SHADOW\_CASTER\_NORMALOFFSET和SHADOW\_CASTER\_FRAGMENT来计算阴影投射，这部分代码也不再粘贴到本书中。

## 17.6 Surface Shader的缺点

从上面的内容中我们可以看出，表面着色器给我们带来了很大的便利。那么，我们之前为什么还要花那么久的时间学习顶点/片元着色器？直接写表面着色器就好了嘛。

正如我们一直强调的那样，表面着色器只是Unity在顶点/片元着色器上面提供的一种封装，是一种更高层的抽象。但任何在表面着色器中完成的事情，我们都可以在顶点/片元着色器中重现。但不幸的是，这句话反过来并不成立。

这世上任何事情都是有代价的，如果我们想要得到便利，就需要以牺牲自由度为代价。表面着色器虽然可以快速实现各种光照效果，但我们失去了对各种优化和各种特效实现的控制。因此，使用表面着色器往往会对性能造成一定的影响，而内置的Shader，例如Diffuse、Bumped Specular等都是使用表面着色器编写的。尽管Unity提供了移动平台的相应版本，例如Mobile/Diffuse和Mobile/Bumped Specular等，但这些版本的Shader往往只是去掉了额外的逐像素Pass、不计算全局光照和其他一些光照计算上的优化。但要想进行更多深层的优化，表面着色器就不能满足我们的需求了。

除了性能比较差以外，表面着色器还无法完成一些自定义的渲染效果，例如10.2.2节中透明玻璃的效果。表面着色器的这些缺点让很多人更愿意使用自由的顶点/片元着色器来实现各种效果，尽管处理光照时这可能难度更大些。

因此，我们给出一些建议供读者参考。

- 如果你需要和各种光源打交道，尤其是想要使用Unity中的全局光照的话，你可能更喜欢使用表面着色器，但要时刻小心它的性能；
- 如果你需要处理的光源数目非常少，例如只有一个平行光，那么使用顶点/片元着色器是一个更好的选择；
- 最重要的是，如果你有很多自定义的渲染效果，那么请选择顶点/片元着色器。

## 第18章 基于物理的渲染

在之前的章节中，我们学习了Lambert光照模型、Phong光照模型和Blinn-Phong光照模型。但这些光照模型的缺点在于，它们都是经验模型。如果我们需要渲染更高质量的画面，这些经验模型就显得不再能满足我们的要求了。

近年来，**基于物理的渲染技术（Physically Based Shading, PBS）**被逐渐应用于实时渲染中。总体来说，PBS是为了对光和材质之间的行为进行更加真实的建模。PBS早已被广泛应用到电影行业中，但游戏中的PBS是近年来才逐渐流行起来的。Unity最早在2012年的《蝴蝶效应》（英文名：Butterfly Effect）的demo中大量使用了PBS，并在Unity 5中正式将PBS引入到引擎渲染中。Unity 5引入了一个名为Standard Shader的可在不同材质之间通用的着色器，而该着色器就是使用了基于物理的光照模型。需要注意的是，PBS并不意味着渲染出来的画面一定是像照片一样真实的，例如，Pixar和Disney尽管长期使用PBS渲染电影画面，但它们得到的风格是非常有特色的艺术风格。相信很多读者或多或少看到过使用PBS渲染出来的画面是多么的酷炫，并很想了解这背后的技术原理。如果你是一个程序员，可能有很大的冲动想要自己实现一个PBS渲染框架，但往往走到后面会发现有很多看不懂的名词以及一大堆与之相关的论文；如果你是一个美工人员，你可能会找到很多关于如何制作PBS中使用的纹理教程，但你大概也了解，想要使用PBS实现出色的渲染效果，并不是纹理+一个Shader这么简单的问题。

现在，我们有一个好消息和一个坏消息要告诉大家。先说好消息，Unity 5引入的基于物理的渲染不需要我们过多地了解PBS是如何实现的，就能利用各种内置工具来实现一个不错的渲染效果。然而坏消息是，我们很难通过短短几万文字来非常详细地告诉读者这些渲染到底是如何实现的，因为这其中需要牵扯许多复杂的光照模型，如果要完全理解每一种模型的话，大概还要讲很多论文和其他参考文献。不过还有一个好消息是，我们相信读者在学完本章后可以了解一些PBS的原理，如果你对PBS有着浓厚的兴趣，想要尝试自己构建一个PBS的渲染框架，可以在本章的扩展阅读部分找到许多非常有价值的参考资料。

在本章中，我们首先会讲解PBS的基本原理，让读者了解它们与我们之前所学的渲染方式到底有哪些不同。尽管本书的定位并不是“教你如何使用Unity”，但我们决定花一点时间来告诉读者Unity 5引入的Standard Shader是如何工作的，以及如何在Unity 5中使用它和其他工具来渲染一个场景，我们希望通过这些内容来让读者明白PBS中的一些关键因素。尽管PBS在手机上的应用并不十分广泛，但我们相信这是未来的发展趋势，希望本章可以为读者打开PBS的大门。

## 18.1 PBS的理论和数学基础

在学习如何实现PBS之前，我们非常有必要来了解基于物理的渲染所基于的理论和数学基础。我们不会过多地牵扯一些论文资料，但如果在阅读过程中读者发现无法理解一些光照模型的实现原理，这可能意味着你需要阅读更多的参考文献。本节主要参考了Naty Hoffman在

SIGGRAPH 2013上做的名为**Background: Physics and Math of Shading**的演讲[1]。

### 18.1.1 光是什么

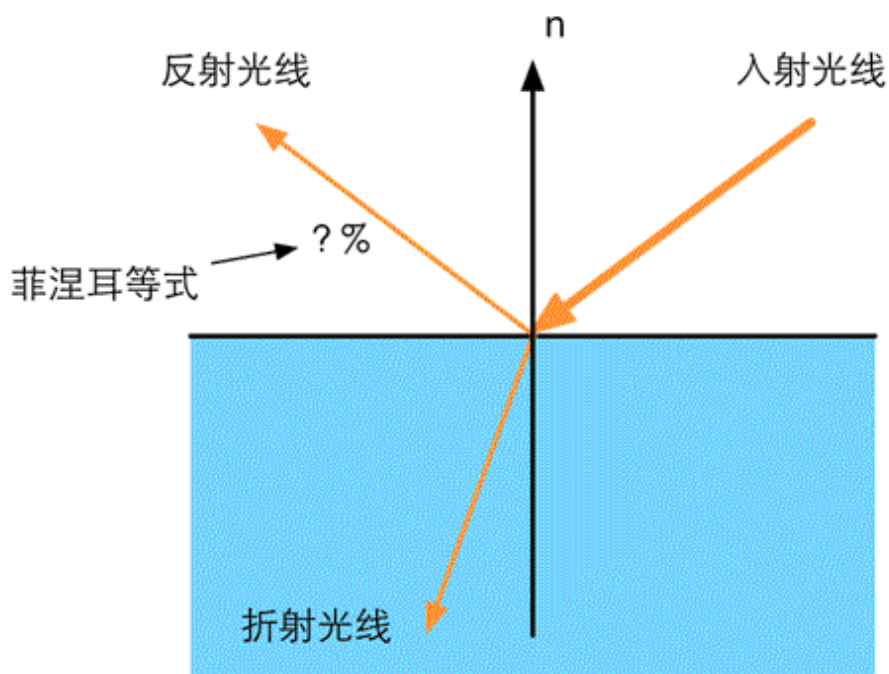
尽管我们之前一直讲光照模型，但要问读者，光到底是什么，可能没有多少人可以解释清楚。在物理学中，光是一种电磁波。首先，光由太阳或其他光源中被发射出来，然后与场景中的对象相交，一些光线被**吸收**（**absorption**），而另一些则被**散射**（**scattering**），最后光线被一个感应器（例如我们的眼睛）吸收成像。

通过上面的过程，我们知道材质和光线相交会发生两种物理现象：散射和吸收（其实还有自发光现象）。光线会被吸收是由于光被转化成了其他能量，但吸收并不会改变光的传播方向。相反的，散射则不会改变光的能量，但会改变它的传播方向。在光的传播过程中，影响光的一个重要的特性是材质的**折射率**（**refractive index**）。我们知道，在均匀的介质中，光是沿直线传播的。但如果光在传播时介质的折射率发生了变化，光的传播方向就会发生变化。特别是，如果折射率是突变的，就会发生光的散射现象。

实际上，在现实生活中，光和物体之间的交互过程是非常复杂的，大多数情况下并不存在一种可分析的解决方法。但为了在渲染中对光照进行建模，我们往往只考虑一种特殊情况，即只考虑两个介质的边界是无限大并且是光学平滑（**optically flat**）的。尽管真实物体的表面并不是无限延伸的，也不是绝对光滑的，但和光的波长相比，它们的大小可以被近似认为是无限大以及光学平滑的。在这样的前提下，光在不同介质的边界会被分割成两个方向：反射方向和折射方



向。而有多少百分比的光会被反射（另一部分就是被折射了）则是由菲涅耳等式（Fresnel equations）来描述的，如图18.1所示。

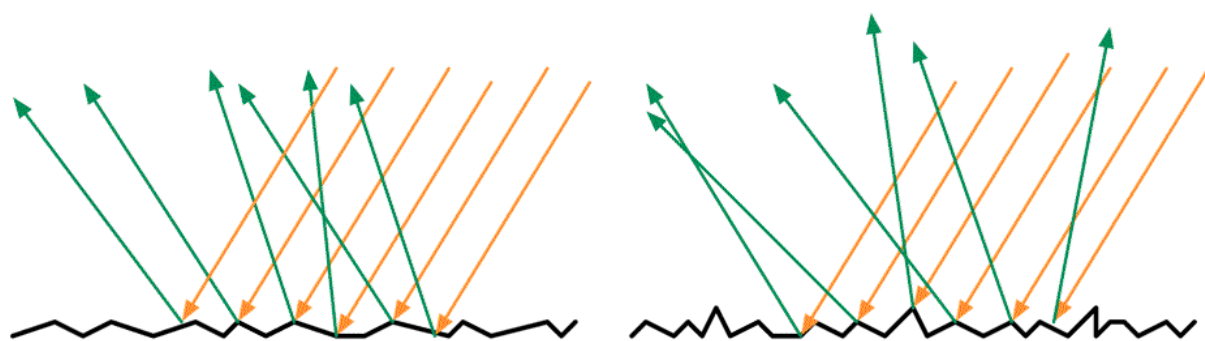


▲图18.1 在理想的边界处，折射率的突变会把光线分成两个方向

但是，这些与光线的交界处真的是像镜子一样平坦吗？尽管在上面我们已经说过，相对于光的波长来说，它们的确可以被认为是光学平坦的。但是，如果想象我们有一个高倍放大镜，去放大这些被照亮的物体表面，就会发现有很多之前肉眼不可见的凹凸不平的平面。在这种情况下，物体的表面和光照发生的各种行为，更像是一系列微小的光学平滑平面和光交互的结果，其中每个小平面会把光分割成不同的方向。

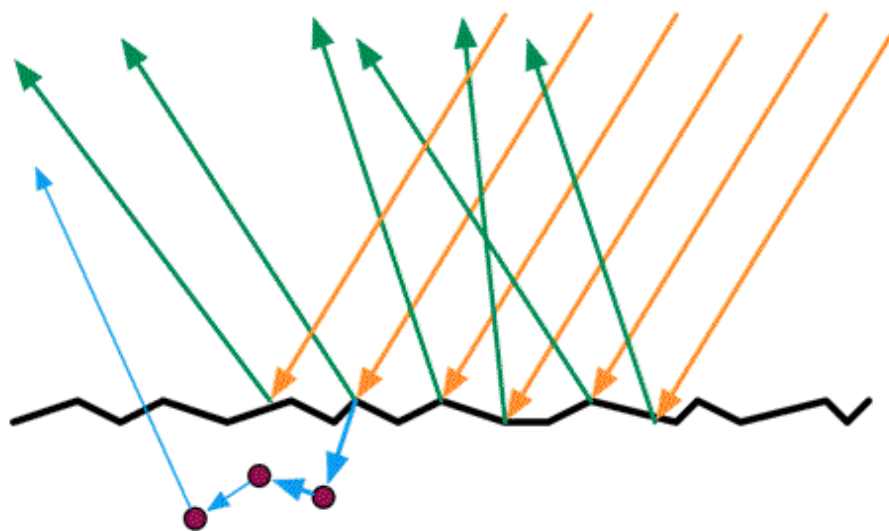
这种建立在微表面的模型更容易解释为什么有些物体看起来粗糙，而有些看起来就平滑，如图18.2所示。想象我们用一个放大镜去观

察一个光滑物体的表面，尽管它的表面仍然由许多凹凸不平的微表面构成，但这些微表面的法线方向变化角度小，因此，由这些表面反射的光线方向变化也比较小，如图18.2左图所示，这使得物体的高光反射更加清晰。而图18.2右图所示的粗糙表面则相反，由此得到的高光反射效果更模糊。



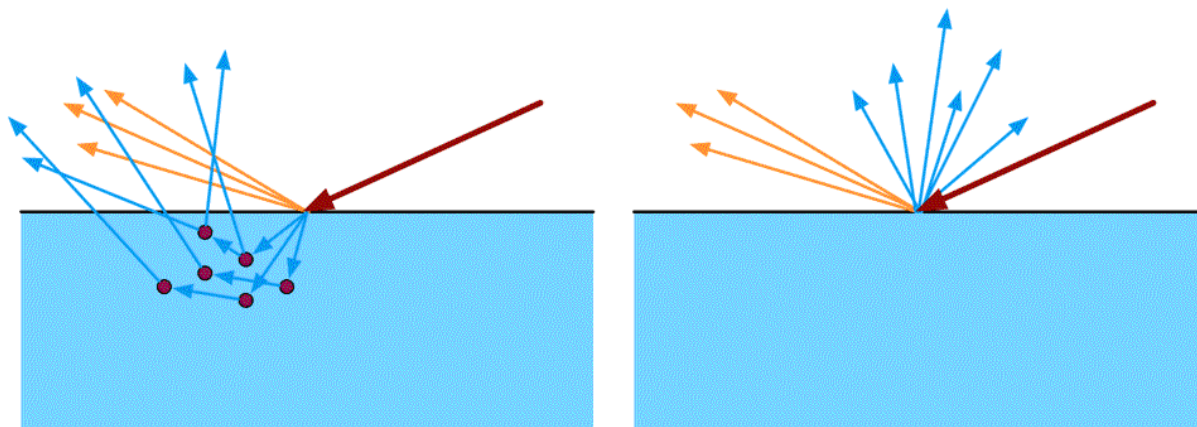
▲ 图18.2 左边：光滑表面的微平面的法线变化较小，反射光线的方向变化也更小。右边：粗糙表面的微平面的法线变化较大，反射光线的方向变化也更大

在上面的内容中，我们并没有讨论那些被微表面折射的光。这些光被折射到物体的内部，一部分被介质吸收，一部分又被散射到外部。金属材料具有很高的吸收系数，因此，所有被折射的光往往会被立刻吸收，被金属内部的自由电子转化成其他形式的能量。而非金属材料则会同时表现出吸收和散射两种现象，这些被散射出去的光又被称为**次表面散射光（subsurface-scattered light）**。在图18.3中，我们给出了一条由微表面折射的光的传播路径（如图18.3所示的蓝线，读者可参考作者给出的彩图）。



▲图18.3 微表面对光的折射。这些被折射的光中 一部分被吸收，一部分又被散射到外部

现在，我们把放大镜从物体表面拿开，继续从渲染的层级大小上考虑光与表面一点的交互行为。那么，由微表面反射的光可以被认为是该点上一些方向变化不大的反射光，如图18.4中的黄线所示。而折射光线（蓝线）则需要更多的考虑。那些次表面散射光会从不同于入射点的位置从物体内部再次射出，如图18.4左图所示。而这些离入射点的距离值和像素大小之间的关系会产生两种建模结果。如果像素要大于这些散射距离的话，意味着这些次表面散射产生的距离可以被忽略，那我们的渲染就可以在局部进行，如图18.4右图所示。如果像素要小于这些散射距离，我们就不可以选择忽略它们了，要实现更真实的次表面散射效果，我们需要使用特殊的渲染模型，也就是所谓的次表面散射渲染技术。



▲图18.4 次表面散射。左边：次表面散射的光线会从不同于入射点的位置射出。如果这些距离值小于需要被着色的像素大小，那么渲染就可以完全在局部完成（右边）。否则，就需要使用次表面散射渲染技术

我们下面的内容均建立在不考虑次表面散射的距离，而完全使用局部着色渲染的前提下。

### 18.1.2 双向反射分布函数（BRDF）

在了解了上面的理论基础后，我们现在来学习如何用数学表达式来表示上面的光照模型。这意味着，我们要对光这个看似抽象的概念进行量化。

我们可以用**辐射率（radiance）**来量化光。辐射率是单位面积、单位方向上光源的辐射通量，通常用 $L$ 来表示，被认为是对单一光线的亮度和颜色评估。在渲染中，我们通常会基于表面的入射光线的入射辐射率 $L_i$ 来计算出射辐射率 $L_o$ ，这个过程也往往被称为是**着色（shading）**过程。

而要得到出射辐射率 $L_o$ ，我们需要知道物体表面一点是如何和光进行交互的。而这个过程就可以使用**BRDF（Bidirectional Reflectance**

**Distribution Function**，中文名称为**双向反射分布函数**）来定量分析。大多数情况下，BRDF可以用 $f(\mathbf{l}, \mathbf{v})$ 来表示，其中 $\mathbf{l}$ 为入射方向和 $\mathbf{v}$ 为观察方向（双向的含义）。这种情况下，绕着表面法线旋转入射方向或观察方向并不会影响BRDF的结果，这种BRDF被称为是**各项同性**（**isotropic**）的BRDF。与之对应的则是**各向异性**（**anisotropic**）的BRDF。

那么，BRDF到底表示的含义是什么呢？BRDF有两种理解方式——第一种理解是，当给定入射角度后，BRDF可以给出所有出射方向上的反射和散射光线的相对分布情况；第二种理解是，当给定观察方向（即出射方向）后，BRDF可以给出从所有入射方向到该出射方向的光线分布。一个更直观的理解是，当一束光线沿着入射方向 $\mathbf{l}$ 到达表面某点时， $f(\mathbf{l}, \mathbf{v})$ 表示了有多少部分的能量被反射到了观察方向 $\mathbf{v}$ 上。

据此，我们给出基于物理渲染的技术中，第一个重要的等式——**反射等式**（**reflection equation**）：

$$L_o(\mathbf{v}) = \int_{\Omega} f(\mathbf{l}, \mathbf{v}) \times L_i(\mathbf{l})(\mathbf{n} \cdot \mathbf{l}) d\omega_i$$

反射等式实际上是渲染方程的一个特殊情况，但它是基于物理基础的。尽管上面的式子看起来有些复杂，但很好理解，即给定观察视角 $\mathbf{v}$ ，该方向上的出射辐射率 $L_o(\mathbf{v})$ 等于所有入射方向的辐射率积分乘以它的BRDF值 $f(\mathbf{l}, \mathbf{v})$ ，再乘以一个余弦值 $(\mathbf{n} \cdot \mathbf{l})$ 。如果积分的概念对某些读者来说难以理解，我们使用更简单的方式来理解。想象我们现在要计算表面上某点的出射辐射率，我们已知到该点的观察方向，该点的出射辐射率是由从许多不同方向的入射辐射率叠加后的

结果。其中，**BRDF**表示了不同方向的入射光在该观察方向上的权重分布。我们把这些不同方向的光辐射率 ( $L_i(\mathbf{l})$ 部分) 乘以观察方向上所占的权重 ( $f(\mathbf{l}, \mathbf{v})$ 部分)，再乘以它们在该表面的投影结果 ( $[(\mathbf{n} \cdot \mathbf{l})]$ 部分)，最后再把这些值加起来（即做积分）就是最后的出射辐射率。

在游戏渲染中，我们通常是和一些**精确光源**（**punctual light sources**）打交道的，而不是计算所有入射光线在半球面上的积分。精确光源指的是那些方向确定、大小为无线小的光源，例如，常见的点光源、聚光灯等。我们使用 $\mathbf{l}_c$ 来表示它的方向，使用 $c_{light}$ 表示它的颜色。使用精确光源的最大的好处在于，我们可以大大简化上面的反射等式。这里省略推导过程（有兴趣的读者可以阅读参考文献[1]），直接给出结论，即对于一个精确光源，我们可以使用下面的等式来计算它在某个观察方向 $\mathbf{v}$ 上的出射辐射率：

$$L_o(\mathbf{v}) = \pi f(\mathbf{l}_c, \mathbf{v}) \times c_{light}(\mathbf{n} \cdot \mathbf{l}_c)$$

和之前使用积分形式的原始反射等式相比，上面的式子使用一个特定的**BRDF**值来代替积分操作，这大大简化了计算。如果场景中包含了多个精确光源，我们可以把它们分别代入上面的式子进行计算，然后把它们的结果相加即可。

下面，我们来看一下反射等式中的重要组成部分——**BRDF**是如何得到的。可以看出，**BRDF**决定了着色过程是否是基于物理的。这可以由**BRDF**是否满足两个特性来判断：它是否满足**交换律**（**reciprocity**）和**能量守恒**（**energy conservation**）。



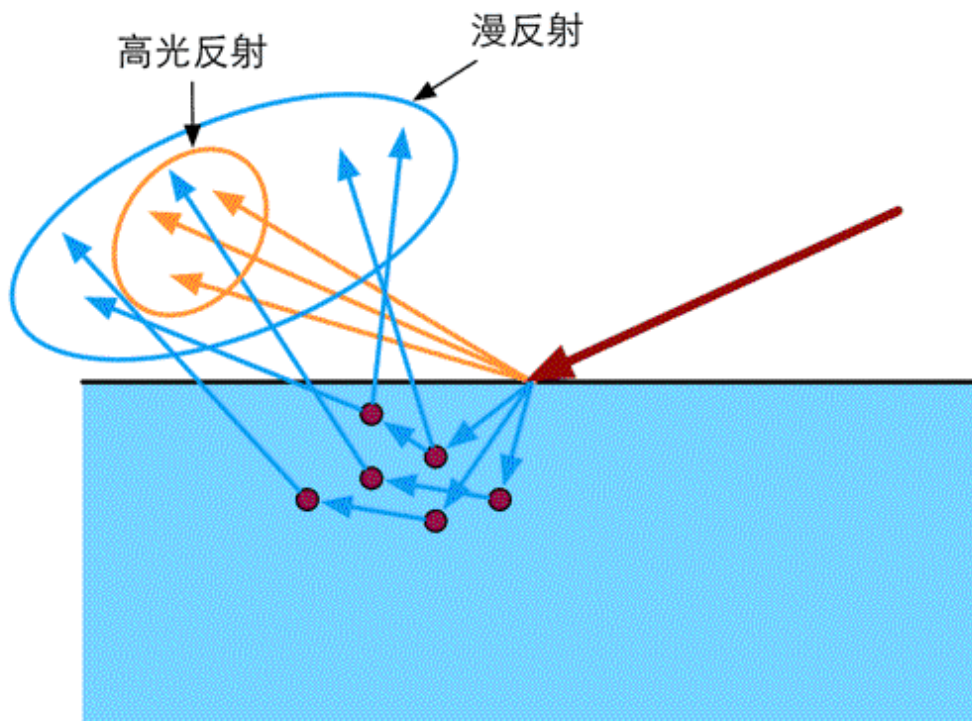
交换律要求当交换 $\mathbf{l}$ 和 $\mathbf{v}$ 的值后，BRDF的值不变，即

$$f(\mathbf{l}, \mathbf{v}) = f(\mathbf{v}, \mathbf{l})$$

而能量守恒则要求表面反射的能量不能超过入射的光能，即

$$\forall \mathbf{l}, \int_{\Omega} f(\mathbf{l}, \mathbf{v})(\mathbf{n} \cdot \mathbf{l}) d\omega_o \leq 1$$

基于这些理论，BRDF可以用于描述两种不同的物理现象：表面反射和次表面散射。针对每种现象，BRDF通常会包含一个单独的部分来描述它们——用于描述表面反射的部分被称为**高光反射项（specular term）**，以及用于描述次表面散射的**漫反射项（diffuse term）**，如图18.5所示。





▲图18.5 BRDF描述的现象。高光反射部分 用于描述反射，漫反射部分用于描述次表面散射

### 18.1.3 漫反射项

我们之前所学习的Lambert模型就是最简单、也是应用最广泛的漫反射BRDF。准确的Lambertian BRDF的表示为：

$$f_{Lambert}(\mathbf{l}, \mathbf{v}) = \frac{c_{diff}}{\pi}$$

其中， $c_{diff}$ 表示漫反射光线所占的比例，它也通常被称为是漫反射颜色（diffuse color）。与我们之前讲过的Lambert光照模型不太一样的，上面的式子实际上是一个定值，我们常见到的余弦（即 $(\mathbf{n} \cdot \mathbf{l})$ ）因子部分实际是反射等式的一部分，而不是BRDF的部分。上面的式子之所以要除以 $\pi$ ，是因为我们假设漫反射在所有方向上的强度都是相同的，而BRDF要求在半球内的积分值为1。因此，给定入射方向 $\mathbf{l}$ 的光源在表面某点的出射漫反射辐射率为：

$$L_{diff} = \frac{c_{diff}}{\pi} \times L_i(\mathbf{l})(\mathbf{n} \cdot \mathbf{l})$$

Lambert模型虽然简单，但很多基于物理的渲染选择使用了更复杂的漫反射项来模拟次表面散射的结果。例如，在Disney使用的BRDF[2]中，它的漫反射项为：

$$f_{diff}(\mathbf{l}, \mathbf{v}) = \frac{baseColor}{\pi} (1 + (F_{D90} - 1)(1 - \mathbf{n} \cdot \mathbf{l})^5)(1 + (F_{D90} - 1)(1 - \mathbf{n} \cdot \mathbf{v})^5)$$

$$\text{其中， } F_{D90} = 0.5 + 2roughness(\mathbf{h} \cdot \mathbf{l})^2$$

在Disney的实现中，*baseColor*是表面颜色，通常由纹理采样得到，*roughness*是表面的粗糙度。上面的漫反射项既考虑了在掠射角（glancing angles）漫反射项的能量变化，还考虑了表面的粗糙度对漫反射的影响。而上面的式子也正是Unity 5内部使用的漫反射项。

#### 18.1.4 高光反射项

在现实生活中，几乎所有的物体都或多或少有高光反射现象。John Hable在他的文章中就强调了**Everything is Shiny**。但在许多传统的Shader中，很多材质只考虑了漫反射效果，而并没有添加高光反射，这使得渲染出来的画面并不那么真实可信。在基于物理的渲染中，BRDF中的高光反射项大多数都是建立在**微面元理论（microfacet theory）**的假设上的。微面元理论认为，物体表面实际是由许多人眼看不到的微面元组成的，虽然物体表面并不是光学平滑的，但这些微面元可以被认为是光学平滑的，也就是说它们具有完美的高光反射。当光线和物体表面一点相交时，实际上是和一系列微面元交互的结果。正如我们在18.1.1节中看到的，当光和这些微面元相交时，光线会被分割成两个方向——反射方向和折射方向。这里我们只需要考虑被反射的光线，而折射光线已经在之前的漫反射项中考虑过了。当然，微面元理论也仅仅是真实世界的散射的一种近似理论，它也有自身的缺陷，仍然有一些材质是无法使用微面元理论来描述的。

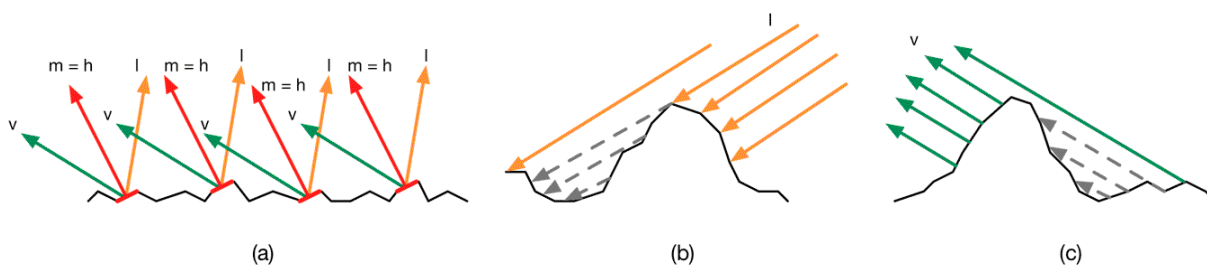
假设表面法线为 $\mathbf{n}$ ，这些微面元的法线 $\mathbf{m}$ 并不都等于 $\mathbf{n}$ ，因此，不同的微面元会把同一入射方向的光线反射到不同的方向上。而当我们计算BRDF时，入射方向 $\mathbf{l}$ 和观察方向 $\mathbf{v}$ 都会被给定，这意味着只有一部分微面元反射的光线才会进入到我们的眼睛中，这部分微面元会恰好把光线反射到方向 $\mathbf{v}$ 上，即它们的法线 $\mathbf{m}$ 等于 $\mathbf{l}$ 和 $\mathbf{v}$ 的一半，也就是我们

一直看到的半角度矢量 $\mathbf{h}$ （half-angle vector，也被称为half vector），如图18.6（a）所示。

然而，这些 $\mathbf{m} = \mathbf{h}$ 的微面元反射也并不会全部添加到BRDF的计算中。这是因为，它们其中一部分会在入射方向 $\mathbf{l}$ 上被其他微面元挡住（shadowing），如图18.6（b）所示，或是在它们的反射方向 $\mathbf{v}$ 上被其他微面元挡住了（masking），如图18.6（c）所示。微面元理论认为，所有这些被遮挡住的微面元不会添加到高光反射项的计算中（实际上它们中的一些由于多次反射仍然会被我们看到，但这不在微面元理论的考虑范围内）。

基于微面元理论的这些假设，BRDF的高光反射项可以用下面的形式来表示：

$$f_{spec}(\mathbf{l}, \mathbf{v}) = \frac{F(\mathbf{l}, \mathbf{h})G(\mathbf{l}, \mathbf{v}, \mathbf{h})D(\mathbf{h})}{4(\mathbf{n} \cdot \mathbf{l})(\mathbf{n} \cdot \mathbf{v})}$$



▲ 图18.6（a） 那些 $\mathbf{m} = \mathbf{h}$ 的微面元会恰好把入射光从 $\mathbf{l}$ 反射到 $\mathbf{v}$ 上，只有这部分微面元才可以添加到BRDF的计算中。（b）一部分满足（a）的微面元会在 $\mathbf{l}$ 方向上被其他微面元遮挡住，它们不会接受到光照，因此会形成阴影。（c）还有一部分满足（a）的微面元会在反射方向 $\mathbf{v}$ 上被其他微面元挡住，因此，这部分反射光也不会被看到

这就是著名的Torrance-Sparrow微面元模型[5]。上面的式子看起来难以理解，实际上其中的各个项对应了我们之前讲到的不同现象。

$D(\mathbf{h})$ 是微面元的**法线分布函数**（**normal distribution function, NDF**），它用于计算有多少比例的微面元的法线满足 $\mathbf{m}=\mathbf{h}$ ，只有这部分微面元才会把光线从 $\mathbf{l}$ 方向反射到 $\mathbf{v}$ 上。 $G(\mathbf{l},\mathbf{v},\mathbf{h})$ 是**阴影—遮掩函数**（**shadowing-masking function**），它用于计算那些满足 $\mathbf{m}=\mathbf{h}$ 的微面元中有多少会由于遮挡而不会被人眼看到，因此它给出了活跃的微面元（**active microfacets**）所占的浓度，只有活跃的微面元才会成功地把光线反射到观察方向上。 $F(\mathbf{l},\mathbf{h})$ 则是这些活跃微面元的**菲涅尔反射**（**Fresnel reflectance**）函数，它可以告诉我们每个活跃的微面元会把多少入射光线反射到观察方向上，即表示了反射光线占入射光线的比率。事实上，现实生活中几乎所有的物体都会表现出菲涅耳现象，读者可以在一篇很有意思的文章**Everything has Fresnel**中看到一些这样的例子。最后，分母 $4(\mathbf{n}\mathbf{l})(\mathbf{n}\mathbf{v})$ 是用于校正从微面元的局部空间到整体宏观表面数量差异的校正因子。

这些不同的部分又可以衍生出很多不同的BRDF实现。例如，我们之前学习的Blinn-Phong模型[7]就是一种非常简单的模型，它使用的法线分布函数 $D(\mathbf{h})$ 为：

$$D_{blinn}(\mathbf{h}) = (\mathbf{n} \cdot \mathbf{h})^{gloss}$$

但实际上Blinn-Phong模型并不能真实地反映很多真实世界中物体的微面元法线反射分布，因此，很多更加复杂的分布函数被提了出来，例如GGX[3]、Beckmann[4]等。同样，阴影-遮掩函数 $G(\mathbf{l},\mathbf{v},\mathbf{h})$ 也有

很多相关工作被提了出来，例如Smith模型[6]。这些数学模型都是为了更加接近使用光学测量仪器测量出来的真实物体的反射分布数据。

尽管存在很多基于物理的BRDF模型，但在真实的电影或游戏制作中，我们希望在直观性和物理可信度之间找到一个平衡点，使得实现的BRDF既可以让美工人员直观地调节各个参数，而又有一定的物理可信度。当然，有时候为了满足直观性我们不得不牺牲一定的物理特性，得到的BRDF可能不是严格基于物理原理的。

在下面的内容中，我们给出Unity 5使用的实现。

### 18.1.5 Unity中的PBS实现

在之前的内容中，我们提到了Unity 5的PBS实际上是受Disney的BRDF[2]的启发。这种BRDF最大的好处之一就是很直观，只需要提供一个万能的Shader就可以让美工人员通过调整少量参数来渲染绝大部分常见的材质。我们可以在Unity内置的UnityStandardBRDF.cginc文件中找到它的实现。

总体来说，Unity 5一共实现了两种PBS模型。一种是基于GGX模型的，另一种则是基于归一化的Blinn—Phong模型的。这两种模型使用了不同的公式来计算高光反射项中的法线分布函数 $D(\mathbf{h})$ 和阴影—遮掩函数 $G(\mathbf{l}, \mathbf{v}, \mathbf{h})$ 。在默认情况下，Unity 5.2使用基于归一化后的Blinn-Phong模型来实现基于物理的渲染（但在Unity 5.3及以后版本中，默认将使用GGX模型，这和很多其他主流引擎的选择一致）。

如前面所讲，Unity使用的BRDF中的漫反射项使用的公式如下：

$$f_{diff}(\mathbf{l}, \mathbf{v}) = \frac{baseColor}{\pi} (1 + (F_{D90} - 1)(1 - \mathbf{n} \cdot \mathbf{l})^5)(1 + (F_{D90} - 1)(1 - \mathbf{n} \cdot \mathbf{v})^5)$$

$$\text{其中, } F_{D90} = 0.5 + 2roughness(\mathbf{h} \cdot \mathbf{l})^2$$

下面我们给出基于GGX模型的高光反射项公式。对于基于归一化的Blinn-Phong模型的高光反射公式，读者可以从UnityStandardBRDF.cginc文件找到它们的实现。

Unity对高光反射项中的法线分布函数 $D(\mathbf{h})$ 采用了GGX模型的一种实现：

$$D_{GGX} = \frac{\alpha^2}{\pi((\alpha^2 - 1)(\mathbf{n} \cdot \mathbf{h})^2 + 1)^2}$$

$$\text{其中, } \alpha = roughness^2$$

阴影-遮掩函数 $G(\mathbf{l}, \mathbf{v}, \mathbf{h})$ 则使用了一种由GGX衍生出的Smith-Schlick模型：

$$G(\mathbf{l}, \mathbf{v}, \mathbf{h}) = \frac{1}{((\mathbf{n} \cdot \mathbf{l})(1 - k) + k)((\mathbf{n} \cdot \mathbf{v})(1 - k) + k)}$$

$$\text{其中, } k = \frac{roughness^2}{2}$$

而菲涅耳反射 $F(\mathbf{l}, \mathbf{h})$ 则使用了图形学中经常使用的Schlick菲涅耳近似等式[7]：

$$F(\mathbf{l}, \mathbf{h}) = F_0 + (1 - F_0)(1 - \mathbf{l} \cdot \mathbf{h})^5$$

其中 $F_0$ 表示高光反射系数，在Unity中往往指的就是高光反射颜色。

上面的公式对于某些读者来说可能晦涩难懂，实际上，这些数学大多来源于对真实世界中各种物体的BRDF的分析，再使用不同的数学模型进行逼近。如果读者想要深入了解基于物理的渲染的数学原理和应用的话，可以参见本章的扩展阅读部分。

幸运的是，在Unity中我们不需要自己在Shader中实现上面的公式，Unity已经为我们提供了现成的基于物理着色的Shader，也就是Standard Shader。

## 18.2 Unity 5的Standard Shader

当我们在Unity 5中新创建一个模型或是新创建一个材质时，其默认使用的着色器都是一个名为Standard的着色器。这个Standard Shader就使用了我们之前所讲的基于物理的渲染。

Unity支持两种流行的基于物理的工作流程：**金属 workflow (Metallic workflow)** 和 **高光反射 workflow (Specular workflow)**。其中，金属 workflow 是默认的工作流程，对应的Shader为Standard Shader。而如果想要使用高光反射 workflow，就需要在材质的Shader下拉框中选择Standard (Specular setup)。需要注意的是，通常来讲，使用不同的 workflow 可以实现相同的效果，只是它们使用的参数不同而已。金属 workflow 并不意味着它只能模拟金属类型的材质，金属 workflow 的名字来源于它定义了材质表面的金属值（是金属类型的还是非金属类型的）。高光反射 workflow 的名字来源于它可以直接指定表面的高光反射颜色（有很强的高



光反射还是很弱的高光反射）等，而在金属工作流中这个颜色需要由漫反射颜色和金属值衍生而来。在实际的游戏制作过程中，我们可以选择自己更偏好的工作流来制作场景，这更多的是个人喜好的问题。当然也可以同时混用两种工作流。

在下面的内容中，我们用Standard Shader来统称Standard和Standard（Specular setup）着色器。Unity提供的Standard Shader允许让我们只使用这一种Shader来为场景中所有的物体进行着色，而不需要考虑它们是否是金属材质还是塑料材质等，从而大大减少我们不断调整材质参数所花费的时间。

### 18.2.1 它们是如何实现的

Standard和Standard（Specular setup）的Shader源代码可以在Unity内置的builtin\_shaders-5.x/ DefaultResourcesExtra文件夹中找到，这些Shader依赖于builtin\_shaders-5.x/CGIncludes文件夹中定义的一些头文件。这些相关的头文件的名称大多类似于UnityStandardXXX.cginc，其中定义了和PBS相关的各个函数、结构体和宏等。表18.1列出了这些头文件的名称以及它们的主要用处。

表18.1

文 件	描 述
UnityPBSLighting.cginc	定义了表面着色器使用的标准光照函数和相关的结构体等，如LightingStandardSpecular函数和SurfaceOutputStandardSpecular结构体

文 件	描 述
UnityStandardCore.cginc	定义了Standard和Standard (Specular setup) Shader使用的顶点/片元着色器和相关的结构体、辅助函数等，如vertForwardBase、fragForwardBase、MetallicSetup、SpecularSetup函数和VertexOutputForwardBase、FragmentCommonData结构体
UnityStandardBRDF.cginc	实现了Unity中基于物理的渲染技术，定义了BRDF1_Unity_PBS、BRDF2_Unity_PBS和BRDF3_Unity_PBS等函数，来实现不同平台下的BRDF
UnityStandardInput.cginc	声明了Standard Shader使用的相关输入，包括shader使用的属性和顶点着色器的输入结构体VertexInput，并定义了基于这些输入的辅助函数，如TexCoords、Albedo、Occlusion、SpecularGloss等函数
UnityStandardUtils.cginc	Standard Shader使用的一些辅助函数，将来可能会移到UnityCG.cginc文件中
UnityStandardConfig.cginc	对Standard Shader的相关配置，例如默认情况下关闭简化版的PBS实现（将UNITY_STANDARD_SIMPLE设为0），以及使用基于归一化的Blinn-Phong模型而非GGX模型来实现BRDF（将UNITY_BRDF_GGX设为0）

文 件	描 述
UnityStandardMeta.cginc	定义了Standard Shader中“LightMode”为“Meta”的Pass（用于提取光照纹理和全局光照的相关信息）使用的顶点/片元着色器，以及它们使用的输入/输出结构体
UnityStandardShadow.cginc	定义了Standard Shader中“LightMode”为“ShadowCaster”的Pass（用于投射阴影）使用的顶点/片元着色器，以及它们使用的输入/输出结构体
UnityGlobalIllumination.cginc	定义了和全局光照相关的函数，如UnityGlobalIllumination函数

我们可以打开Standard.shader和StandardSpecular.shader文件来分析Unity是如何实现基于物理的渲染的。总体来讲，这两个Shader的代码基本相同——它们都定义了两个SubShader，第一个SubShader使用的计算更加复杂，主要针对非移动平台（通过`#pragma exclude_renderers gles`代码来排除GLES平台），并定义了前向渲染路径和延迟渲染路径使用的Pass，以及用于投射阴影和提取元数据的Pass；第二个SubShader定义了4个Pass，其中两个Pass用于前向渲染路径，一个Pass用于投射阴影，另一个Pass用于提取元数据，该SubShader主要针对移动平台。Standard.shader和StandardSpecular.shader最大的不同之处在于，它们在设置BRDF的输入时使用了不同的函数来设置各个参数——基于金属工作流的Standard Shader使用MetallicSetup函数来设置各个参数，基于高光反射工作流的Standard（Specular setup）Shader使用

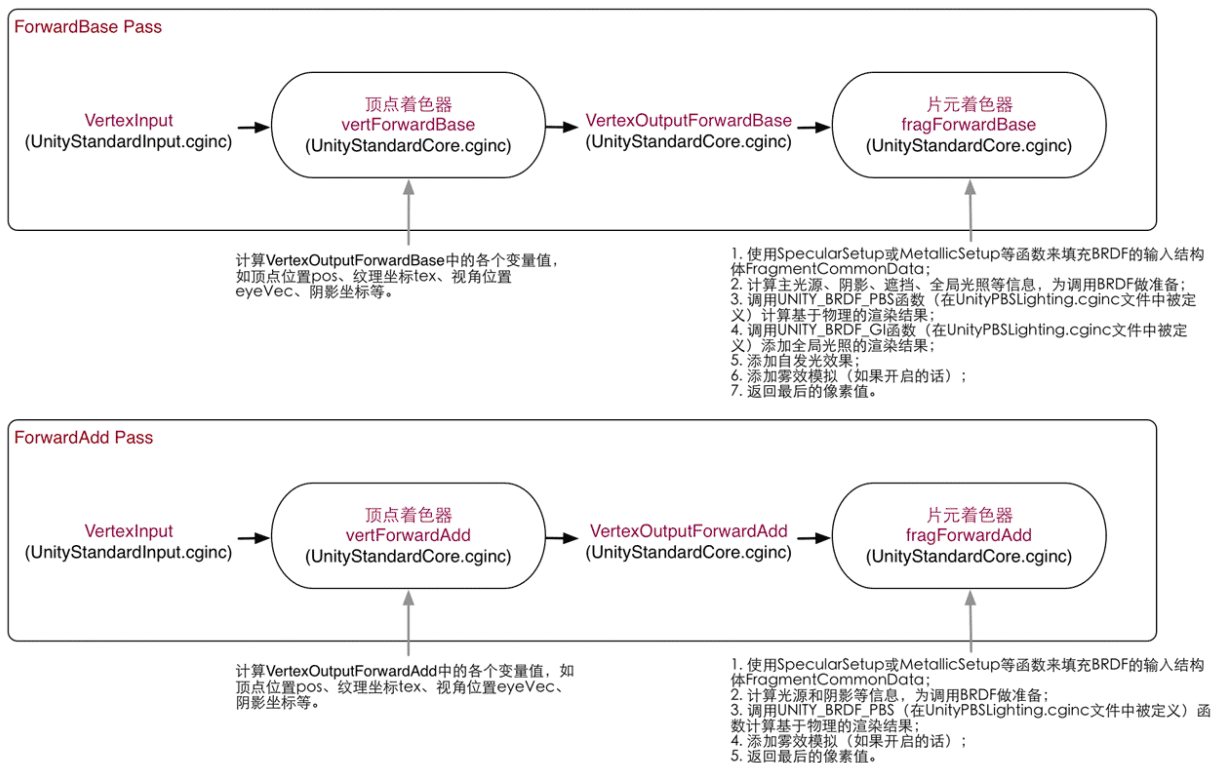
SpecularSetup函数来设置。MetallicSetup和SpecularSetup函数均在UnityStandardCore.cginc文件中被定义。图18.7给出了Standard Shader中用于前向渲染路径的典型实现，这是由对内置文件的分析所得。

从图18.7中可以看出，两个Pass的代码大体相同，只是ForwardBase Pass进行了更多的光照计算，例如，计算全局光照、自发光等效果，这些计算只需要在物体的整个渲染过程中计算一次即可，因此不需要在ForwardAdd Pass中再计算一次，这与我们之前学习前向渲染时的经验一致。

### 18.2.2 如何使用Standard Shader

我们之前提到，Unity 5的Standard Shader适用于各种材质的物体，但是，我们应该如何使用Standard Shader来得到不同的材质效果呢？

我们首先来回答一个问题，为什么不同的材质看起来是如此不同呢？这需要回顾我们在18.1节讲到的内容。我们知道，材质和光的交互可以分成漫反射和高光反射两个部分，其中漫反射对应了次表面散射的结果，而高光反射则对应了表面反射的结果。通过对金属材质和非金属材质的分析，我们可以得到它们的漫反射和高光反射的一些特点。



▲ 图18.7 Standard Shader中前向渲染路径使用的Pass（简化版本的PBS使用了VertexOutputBaseSimple等结构体来代替相应的结构体）

## 1. 金属材料

- 几乎没有漫反射，因为所有被吸收的光都会被自由电子立刻转化为其他形式的能量；
- 有非常强烈的高光反射；
- 高光反射通常是有颜色的，例如金子的反光颜色为黄色。

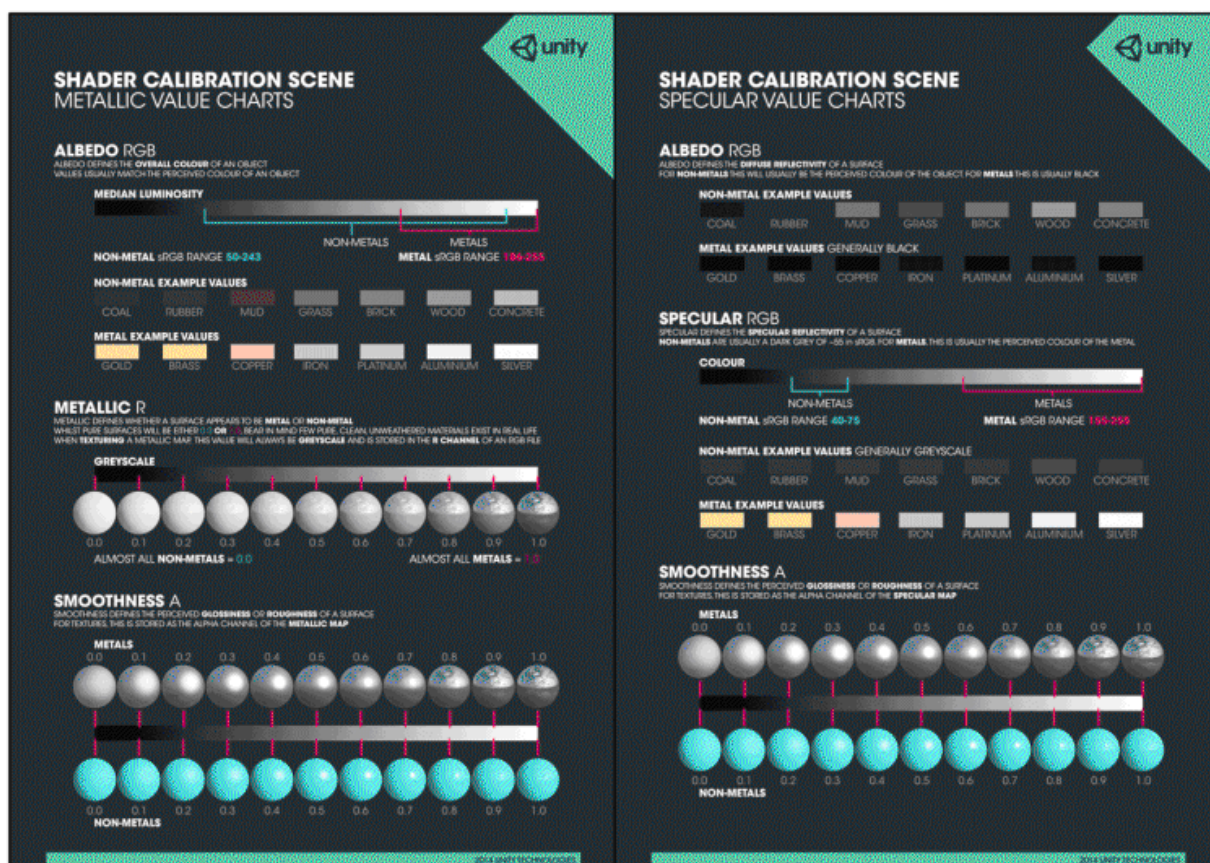
## 2. 非金属材料

- 大多数角度高光反射的强度比较弱，但在掠射角时高光反射强度反而会增强，即菲涅耳现象；
- 高光反射的颜色比较单一；



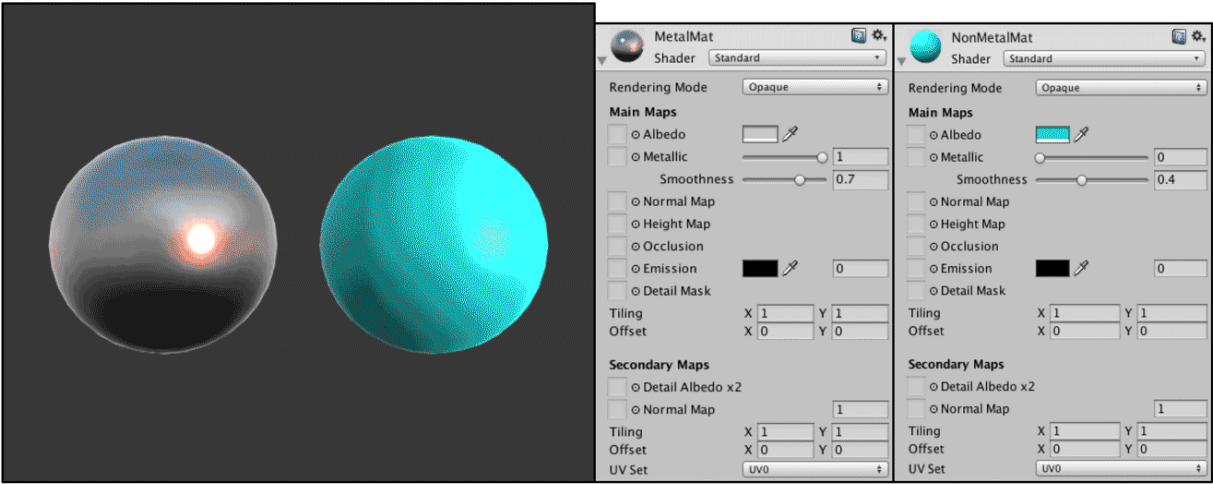
- 漫反射的颜色多种多样。

但真实的材质大多混合了上面的这些特性，Unity提供的工作流就是为了更加方便地让我们针对以上特性来调整材质效果。在Unity官方提供的示例项目**Shader Calibration Scene** (<https://www.assetstore.unity3d.com/en/#!/content/25422>) 中，Unity提供了两个非常有参考价值的校准表格，如图18.8所示，它们分别对应了金属工作流和高光反射工作流使用的参考属性值，来方便我们针对不同类型的材质来调整参数。读者也可以在本书资源的Assets/Textures/Chapter18/Charts文件夹找到这两张校准表格。



▲ 图18.8 Unity提供的校准表格。左边：金属工作流 使用的校准表格，右边：高光反射工作流使用的校准表格

我们以图18.8的左图，即金属工作流使用的校准表格为例，来解释如何使用这张校准表格来指导我们调整材质。在本书资源的场景文件Scene\_18\_2中，我们提供了一个简单的场景来展示不同材质的结果。图18.9显示了场景结果以及物体使用的材质。需要注意的是，读者需要在Edit → Project Settings → Player → Other Settings → Color Space中选择Linear才可以得到和图18.9中相同的效果，这是因为基于物理的渲染需要使用线性空间（详见18.3.4节）来进行相关计算。



▲图18.9 使用金属工作流来实现不同类型的材质。左边的球体：金属材质，右边的球体：塑料材质

在金属工作流中，材质面板中的**Albedo**定义了物体的整体颜色，它通常就是我们视觉上认为的物体颜色。从亮度来看，非金属材质的亮度范围通常在50~243，而金属材质的亮度一般在186 255之间。Unity给的校准表格（见图18.8中的左图）中还给出了一些非金属材质和金属材质使用的示例**Albedo**属性值，我们可以直接使用这些示例值来作为材质属性。当然，也可以直接使用一张纹理作为材质的**Albedo**值。在我们的例子中，我们把金属材质（图18.9中的左边的球体）的



**Albedo**设为银灰色，而把塑料材质（图18.9中的右边的球体）的设为蓝绿色。材质面板中的下一个属性是**Metallic**，它定义了该物体表面看起来是否更像金属或非金属。同样，我们也可以使用一张纹理来采样得到表面的**Metallic**值，此时该纹理中的R通道值将对应了**Metallic**值。在我们的例子中，我们把金属材质的**Metallic**值设为1，表明该物体几乎完全是一个金属材质，同时把塑料材质的**Metallic**值设为0，表明该物体几乎没有任何金属特性。最后一个重要的材质属性是**Smoothness**，它是上一个属性**Metallic**的附属值，定义了从视觉上来看该表面的光滑程度。如果我们在设置**Metallic**属性时使用的是一张纹理，那么这张纹理的A通道就对应了表面的**Smoothness**值（此时纹理的GB通道则被忽略）。在我们的例子中，我们把金属材质的**Smoothness**值设置为相对较大的0.7，表明该金属表面比较光滑，而把塑料材质的**Smoothness**值设为0.4，表明该塑料表明比较粗糙。

高光反射工作流使用的面板和上述金属工作流使用的基本相同，只是使用了不同含义的**Albedo**属性，并使用**Specular**代替了上述的**Metallic**属性。在高光反射工作流中，材质的**Albedo**属性定义了表面的漫反射强度。对于非金属材质，它的值通常仍然是视觉上认为的物体颜色，但对于金属材质，**Albedo**的值通常非常接近黑色（还记得吗，金属材质几乎不存在次表面散射的现象）。高光反射工作流的**Specular**属性则定义了表面的高光反射强度。非金属材质通常使用一个灰度值范围在0~55的深灰色来作为**Specular**值，表明非金属材质的高光反射较弱。金属材质则通常会使用视觉上认为的该金属的颜色作为它的**Specular**值。**Specular**属性同样也有一个子属性**Smoothness**，它定义了从视觉上来看该表面的光滑程度。和上面的金属工作流类似，如果使

用了一张纹理来为**Specular**属性赋值，那么纹理的RGB通道对应了**Specular**属性值，A通道对应了**Smoothness**属性值。

上述材质属性都属于材质面板中的**Main Maps**部分，除了上述提到的属性外，**Main Maps**还包含了其他材质属性，例如，切线空间下的法线纹理、遮挡纹理、自发光纹理等。**Main Maps**部分的下面还有一个**Secondary Maps**的属性部分，这个部分的属性是用来定义额外的细节信息，这些细节通常会直接绘制在**Main Maps**的上面，来为材质提供更多的微表面或细节表现。

除了上述属性，我们还可以为**Standard Shader**选择它使用的渲染模式，即材质面板上的**Render Mode**选项。**Standard Shader**支持4种渲染模式，分别是**Opaque**、**Cutout**、**Fade**和**Transparent**。其中，**Opaque**用于渲染最常见的不透明物体，这也是默认的渲染模式。对于像玻璃这样的材质，我们可以选择**Transparent**模式，在这个渲染模式下，**Albedo**属性的A通道用于控制材质的透明度。而在**Cutout**渲染模式下，**Albedo**属性中纹理的A通道会成为一个掩码纹理，而它的子属性**Alpha Cutoff**将是透明度测试时使用的阈值。**Fade**模式和**Transparent**模式是类似的，不同的是，在**Transparent**模式下，当材质的透明值不断降低时，它的反射仍然能被保留，而在**Fade**模式下，该材质的所有渲染效果都会逐渐从屏幕上淡出。

需要注意的是，尽管**Standard Shader**的材质面板有许多可供调节的属性，但我们不用担心由于没有使用一些属性而会对性能有所影响。**Unity**在背后已经进行了高度优化，在我们生成可执行程序时，**Unity**会

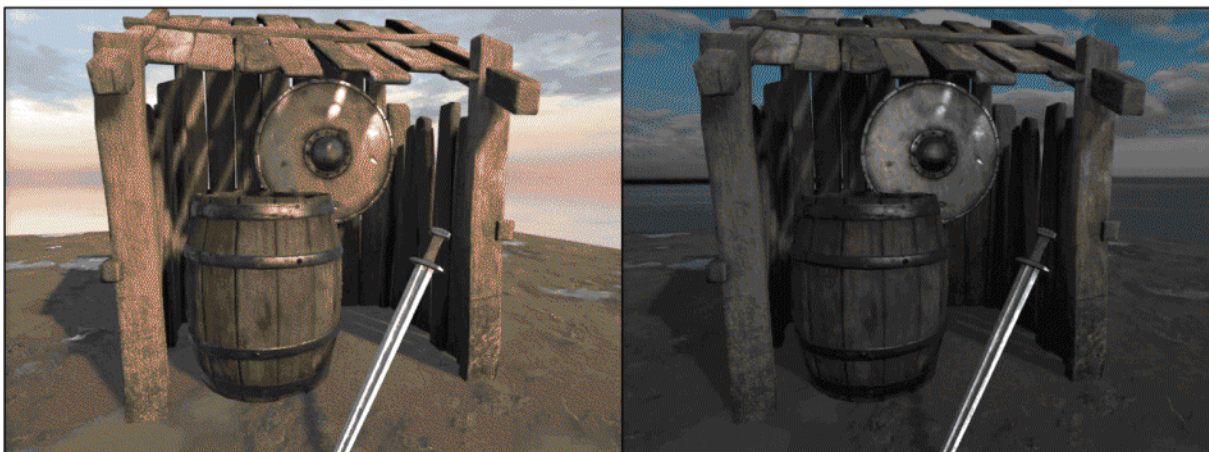
检查哪些属性没有被使用到，同时也会针对目标平台进行相应的优化。

从上面的内容可以看出，要想得到可信度更高的渲染结果，我们需要对不同材质使用合适的属性值，尤其是一些重要的属性值，例如 **Albedo**、**Metallic** 和 **Specular**。当然，想要让整个场景的渲染结果令人满意，尤其包含了复杂光照的场景，仅仅有这些使用了 **PBS** 的材质是不够的，我们需要使用 Unity 提供的其他一些重要的技术，例如 **HDR** 格式的 **Skybox**、全局光照、反射探针、光照探针、**HDR** 和屏幕后处理等。

### 18.3 一个更加复杂的例子

在本章最后，我们将以一个更加复杂的、基于物理渲染的场景结束，该场景对应了本书资源中的 **Scene\_18\_3**。本场景使用的元素大多来源于 Unity 官方的示例项目 **Viking Village** (<https://www.assetstore.unity3d.com/jp/#!/content/29140>)，读者可以下载完整的项目来更加深入地学习 Unity 中的 **PBS**。

图 18.10 展示了在不同光照条件下本例实现的效果。需要注意的是，读者需要在 **Edit** → **Project Settings** → **Player** → **Color Space** 中选择 **Linear** 才可以得到和图 18.9 中相同的效果，这是因为基于物理的渲染需要使用线性空间（详见 18.3.4 节）来进行相关计算。



▲ 图18.10 在Unity 5中使用基于物理的渲染技术，场景在不同光照下的渲染结果

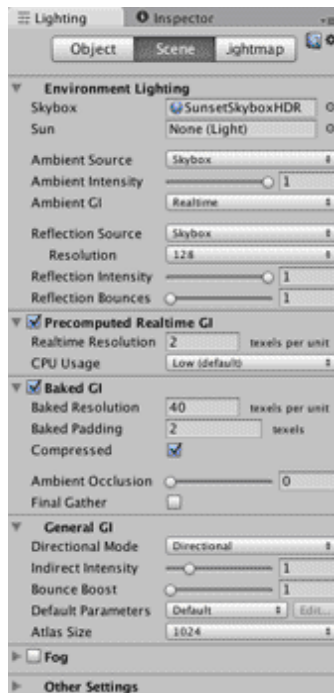
那么，基于物理的Standard Shader是如何与其他Unity功能相互配合得到这样的场景呢？

### 18.3.1 设置光照环境

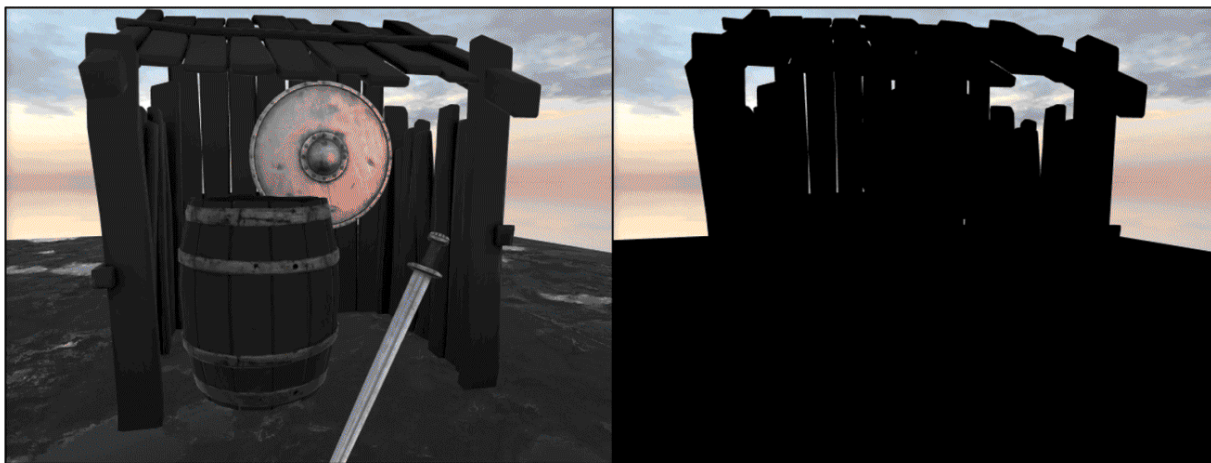
我们首先需要为场景设置光照环境。在默认情况下，Unity 5中一个新创建的场景会包含一个默认的Skybox。在本例中，我们使用一个自定义的Skybox来代替默认值。做法是，打开Window → Lighting，在Scene标签页下把本例使用的SunsetSkyboxHDR拖曳到Skybox选项中，如图18.11所示。

本例中的Skybox使用了一个HDR格式的Cubemap，这与我们之前在10.1节中制作Skybox时使用的纹理不同。这需要解释HDR（High Dynamic Range）的相关知识，我们将在18.4.3节更加详细地介绍HDR的原理和应用。但在这里，我们只需要知道，使用HDR格式的Skybox可以让场景中物体的反射更加真实，有利于我们得到更加可信的光照效果。

我们还可以设置场景使用的**环境光照**，这些环境光照可以对场景中所有的物体表面产生影响。在图18.11所示的设置面板中，我们可以选择环境光照的来源（**Ambient Source**选项），是来自于场景使用的Skybox，还是使用渐变值，亦或是某个固定的颜色。我们还可以设置环境光照的强度（**Ambient Intensity**参数），如果想要场景中的所有物体不接受任何环境光照，可以把该值设为0。在使用了Standard Shader的前提下，如果我们关闭场景中所有的光源，并把环境光照的强度设为0，场景中的物体仍然可以接受一些光照，如图18.12中的左图所示。



▲ 图18.11 光照面板下的Scene标签页



▲ 图18.12 左边：当关闭场景中的所有光源并把环境光照强度设为0后，使用了Standard Shader的物体仍然具有光照效果，右边：在左图的基础上，把反射源设置为空，使得物体不接受任何默认反射信息

那么，这些光照是从哪里来的呢？答案就是**反射**。默认的反射源（**Reflection Source**选项）是场景使用的Skybox。如果我们不想让场景中的物体接受任何默认的反射光照，可以把反射源设置为自定义（即**Custom**），并把自定义的Cubemap保留为空即可（另一种方式是直接把场景使用的Skybox设置为空），如图18.12右图所示。但为了得到更加逼真的渲染结果，我们通常是不会这样做的。在渲染实现上，即便场景中没有任何光源，Unity在内部仍然会调用**ForwardBase Pass**（假设使用的是前向渲染路径的话），并使用反射的光照信息来填充光源信息，再进行基于物理的渲染计算。读者可以通过帧调试器（**Frame Debugger**）来查看渲染过程。需要注意的是，这里设置的反射源是默认的反射源，如果我们在场景中添加了其他反射探针（**Reflection Probes**，见18.3.2节），物体可能会使用其他反射源。当默认反射源是Skybox时，Unity会由场景使用的Skybox生成一个Cubemap，我们可以通过**Resolution**选项来控制它每个面的分辨率。



除了Standard Shader外，Unity还引入了一个重要的流水线——实时**全局光照（Global Illumination，GI）**流水线。使用GI，场景中的物体不仅可以受直接光照的影响，还可以接受间接光照的影响。直接光照指的是那些直接把光照射到物体表面的光源，在本书之前的章节中，我们使用的都是直接光照来渲染场景中的物体。但在现实生活中，物体还会受到间接光照的影响。例如，想象一个红色墙壁旁边放置了一个球体，尽管墙壁本身不发光，但球体靠近墙的一面仍会有少许的红色，这是由于红色墙壁把一些间接光照投射到了球体上。在Unity中，间接光照指的就是那些被场景中其他物体反弹的光，这些间接光照会受反弹光的表面的颜色影响（例如之前例子中的红色的墙壁），这些表面会在反弹光线时把自身表面的颜色添加到反射光的计算中。在Unity 5中，我们可以使用这些直接光照和间接光照来创建更加真实的视觉效果。

下面，我们首先设置场景使用的**直接光照**——一个平行光。在PBR（Physically Based Rendering）中，想要让渲染效果更加真实可信，我们需要保证平行光的方向和Skybox中的太阳或其他光源的位置一致，使得物体产生的光照信息可以与Skybox互相吻合。有时，我们可能会使用一张耀斑纹理（Flare Texture）来模拟太阳等光源，此时我们同样需要确保平行光的方向与耀斑纹理的位置一致。与之类似的还有平行光的颜色，我们应该尽量让平行光的颜色和场景环境相匹配。例如，在图18.10的左图中，场景的光照环境为日落时分，因此平行光的颜色为浅黄色，如图18.13所示，而在图18.10的右图中，场景的光照环境更接近傍晚，此时平行光的颜色为淡蓝色。我们还在Skybox的材质面板上调整天空的旋转角度及曝光度，来调整场景的背景。

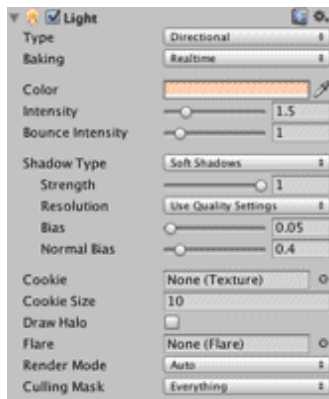


在平行光面板的烘焙选项（即**Baking**）中，我们选择了**Realtime**模式，这意味着，场景中受平行光影响的所有物体都会进行实时的光照计算，当光源或场景中其他物体的位置、旋转角度等发生变化时，场景中的光照结果也会随之变化。然而，实时光照往往需要较大的性能消耗，对于移动平台这样资源比较短缺的平台，我们可以选择**Baked**模式，此时，Unity会把该光源的光照效果烘焙到一张光照纹理

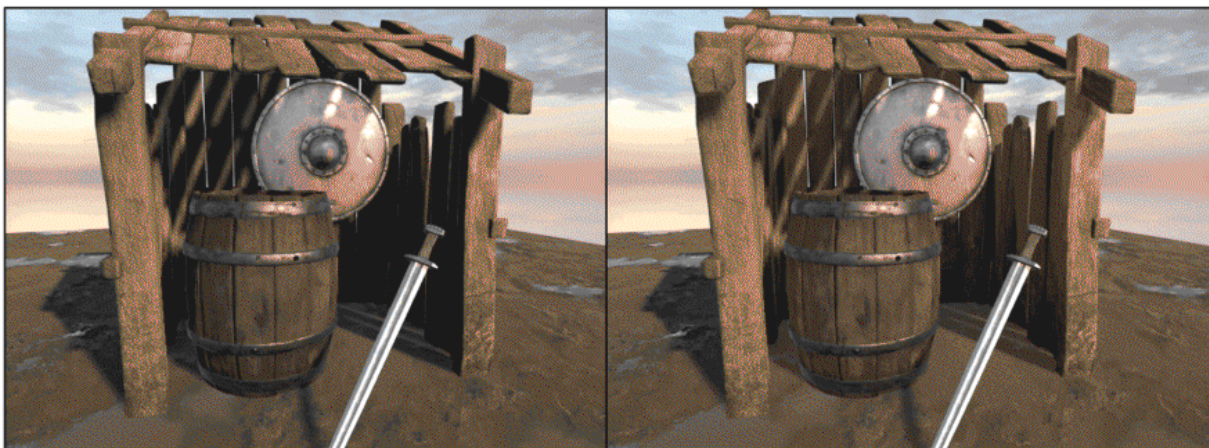
（**lightmap**）中，这样我们就不用实时为物体计算复杂的光照，而只需要通过纹理采样来得到光照结果。选择烘焙模式的缺点在于，如果场景中的物体发生了移动，但是它的阴影等光照效果并不会发生变化。烘焙选项中的**Mix**模式则允许我们混合使用实时模式和烘焙模式，它会把场景中的静态物体（即那些被标识为**Static**的物体）的光照烘焙到光照纹理中，但仍然会对动态物体产生实时光照。

Unity 5引入了实时间接光照的功能，在这个系统下，场景中的直接光照会在场景中各个物体之间来回反射，产生**间接光照**。正如我们之前讲到的，间接光照可以让那些没有直接被光源照亮的物体同样可以接受到一定的光照信息，这些光照是由它周围的物体反射到它的表面上的。当一条光线从光源被发射出来后，它会与场景中的一些物体相交，第一个和光线相交的物体受到的光照即为直接光照。当得到直接光照在该物体上的光照结果后，该物体还会继续反射该光线，从而对其他物体产生间接光照。此后与该光线相交的物体，就会受到间接光照的影响，同时它们也会继续反射。当经过多次反射后，该光线最后完全消失。这些间接光照的强度是由GI系统计算得到的默认亮度值。图18.13所示的光源面板中的**Bounce Intensity**参数可以让我们调节这些间接光照的强度。当我们把它设为0时，意味着一条光线仅会和一个物体相交，不再被继续反射，也就是说，场景中的物体只会受到直

接光照的影响。图18.14显示了Bounce Intensity分别为0和8时，场景的渲染结果，注意其中阴影部分的细节。



▲ 图18.13 使用的平行光



▲ 图18.14 左边：将Bounce Intensity设置为0，物体不再受到间接光照的影响，木屋内阴影部分的可见细节很少。右边：将Bounce Intensity设为8，阴影部分的细节更加清楚

除了上述调整单个光源的间接光照强度，我们也可以对整个场景的间接光照强度进行调整。这是按照图18.11所示的光照面板来实现的。在光照面板的Scene标签页下，我们可以调整General GI参数块中的Bounce Boost参数来控制场景中反射的间接光照的强度，它会和单个光源的Bounce Intensity参数来一起控制间接光照的反射强度。除此之外，

把Indirect Intensity参数调大同样可以增大间接光照的强度。需要注意的是，间接光照还有可能来自一些自发光的物体。

### 18.3.2 放置反射探针

回忆我们在10.1节中讲到的环境映射，在实时渲染中，我们经常会使用Cubemap来模拟物体的反射效果。例如，在赛车游戏中，我们需要对车身或车窗使用反射映射的技术来模拟它们的反光材质。然而，如果我们永远使用同一个Cubemap，那么，当赛车周围的场景发生较大变化时，就容易出现“穿帮镜头”，因为车身或车窗的环境反射并没有随着环境变化而发生变化。一种解决办法是可以在脚本中控制何时生成从当前位置观察到的Cubemap，而Unity 5为我们提供了一种更加方便的途径，即使用**反射探针（Reflection Probes）**。反射探针的工作原理和光照探针（Light Probes）类似，它允许我们在场景中的特定位置对整个场景的环境反射进行采样，并把采样结果存储在每个探针上。当游戏中包含反射效果的物体从这些探针附近经过时，Unity会把从这些邻近探针存储的反射结果传递给物体使用的反射纹理。如果物体周围存在多个反射探针，Unity还会在这些反射结果之间进行插值，来得到平滑渐变的反射效果。实际上，Unity会在场景中放置一个默认的反射探针，这个反射探针存储了对场景使用的Skybox的反射结果，来作为场景的环境光照（见18.3.1节）。如果我们需要让场景中的物体包含额外的反射效果，就需要放置更多的反射探针。

反射探针同样有 3 种类型：**Baked**，这种类型的反射探针是通过提前烘焙来得到该位置使用的Cubemap的，在游戏运行时反射探针中存储的Cubemap并不会发生变化。需要注意的是，这种类型的反射探针在烘焙时同样只会处理那些静态物体（即那些被标识为Reflection Probe

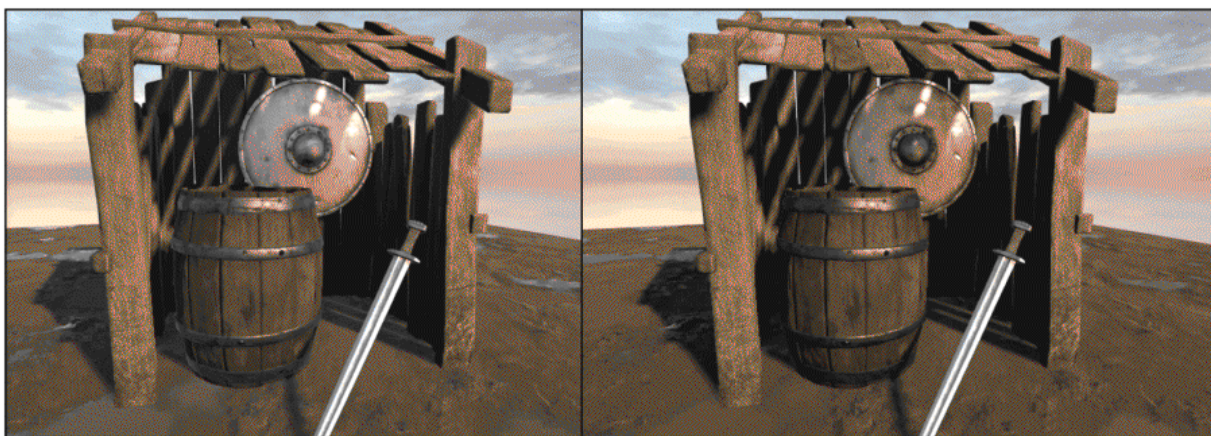
Static的物体)；**Realtime**，这种类型则会实时更新当前的Cubemap，并且不受静态物体还是动态物体的影响。当然，这种类型的反射探针需要花费更多的处理时间，因此，在使用时应当非常小心它们的性能。幸运的是，**Unity**允许我们从脚本中通过触发来精确控制反射探针的更新；最后一种类型是**Custom**，这种类型的探针既可以让从编辑器中烘焙它，也可以让我们使用一个自定义的Cubemap来作为反射映射，但自定义的Cubemap不会被实时更新。

我们在本节使用的场景中放置了3个反射探针，它们的类型都是**Baked**（前提是我们把场景中的物体标识成了**Static**）。使用反射探针前后的对比效果如图18.15所示。

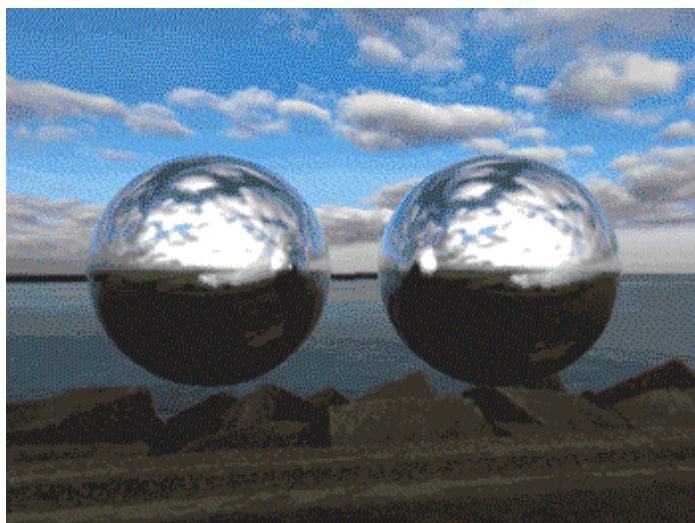
需要注意的是，在放置反射探针时，我们选取的位置并不是任意的。通常来说，反射探针应该被放置在那些具有明显反射现象的物体的旁边，或是一些墙角等容易发生遮挡的物体周围。在本例使用的场景中，木屋内的盾牌具有比较明显的反射效果，而盾牌本身又被木屋遮挡，因此，其中一个反射探针的位置就在盾牌附近。当我们放置好探针后，我们还需要为它们定义每个探针的影响区域，当反射物体进入到这个区域后，反射探针就会对物体的反射产生影响。通常情况下，反射探针的影响区域之间往往会有所重叠，例如，本例中盾牌附近的反射探针和另外两个（一个在木屋前方，一个在木屋后方）的影响区域都有所重叠。此时，**Unity**会计算反射物体的包围盒与这些重叠区域的交叉部分，并据此来选择使用的反射映射。如果当前的目标平台使用的是**SM 3.0**及以上的话，**Unity**还可以允许我们在这些互相重叠的反射探针之间进行混合，来实现平缓的反射过渡效果。



使用Unity内置的反射探针的另一个好处是，我们可以模拟**互相反射（interreflections）**。我们曾在10.1节中讲到使用传统的Cubemap方法无法模拟互相反射的效果。例如，假设场景中有两面互相面对面的镜子，在理想情况下，它们不仅会反射自己对面的那面镜子，也会反射那面镜子里反射的图像。只要反射光线没有被完全吸收，反射就会一直进行下去。要实现这种效果，就需要追踪光线的反射轨迹，这是传统的反射方法所无法实现的。Unity 5引入的GI系统让这种效果变成了可能，我们在本书资源的Scene\_18\_3\_2场景中展示了这样的一个例子，如图18.16所示。我们可以在图18.16中看到，两个金属反射的图像包含了两次互相反射的效果。



▲ 图18.15 左边：未使用反射探针。右边：在场景中放置了两个反射探针，注意墙上的盾牌与左图的差别



▲图18.16 使用反射探针实现相互反射的效果

在图18.16所示的场景中，我们在每个金属球的位置处放置了一个反射探针，并把每个金属球上的**Mesh Renderer**组件中的**Reflection Probes**设置为**Simple**，这样保证它们只会使用离它们最近的一个反射探针。默认情况下，反射探针只会捕捉一次反射，也就是说，左边金属球使用的反射探针只会捕捉到由右边的金属球第一次反射过来的光线。但在理想情况下，反射过来的光线会继续被左边的金属球反射，并对右边的金属球造成影响。**Unity**允许我们控制物体之间这样来回反射的次数，这可以通过改变图18.11中的**Reflection Bounces**参数来实现。在图18.16所示的场景中，我们把该值设为了2。

然而，正如本节一开始所提到的，使用反射探针往往会需要更多的计算时间。这些探针实际上也是通过在它的位置上放置一个摄像机，来渲染得到一个**Cubemap**。如果我们把反弹次数设置的很大，或是使用实时渲染，那么这些探针很可能会造成性能瓶颈。更多关于如何优化反射探针以及它的高级用法，读者可以参见**Unity**的官方手册（<http://docs.unity3d.com/Manual/ReflectionProbes.html>）。

### 18.3.3 调整材质

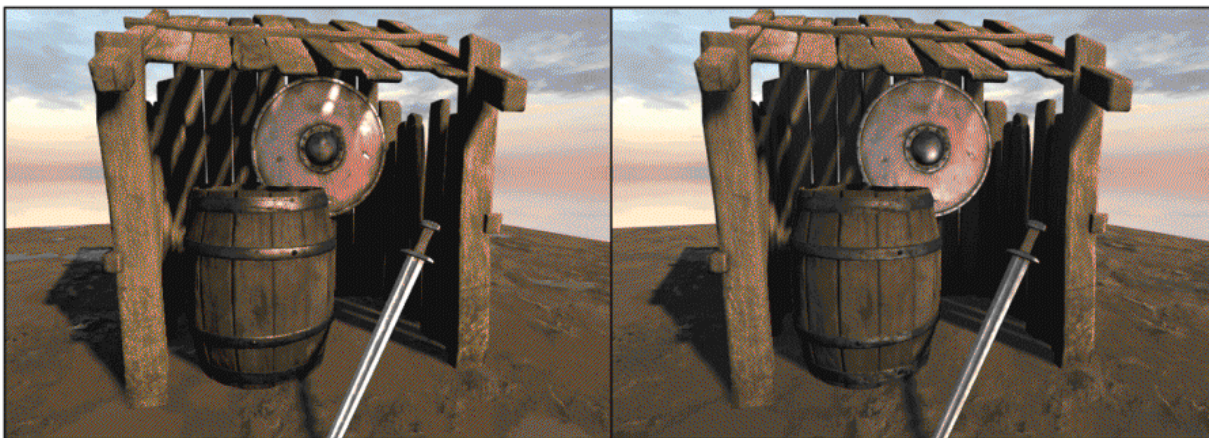
要得到真实可信的渲染效果，我们需要为场景中的物体指定合适的材质。需要再次提醒读者的是，基于物理的渲染并不意味着一定要模拟像照片真实的效果。基于物理的渲染更多的好处在于，可以让我们的场景在各种光照条件下都能得到令人满意的效果，同时不需要频繁地调整材质参数。

在Unity中，要想和全局光照、反射探针等内置功能良好地配合来得到出色的渲染结果，就需要使用Unity内置的Standard Shader。我们已经在18.2.2节中学习了如何针对不同类别的物体来调整它们使用的材质属性。在本例中，我们使用了更复杂的纹理和模型，它们都来自于Unity官方的示例项目**Viking Village**。这些材质可以为读者制作自己的材质提供一些参考，例如，场景中所有物体都使用了高光反射纹理（Specular Texture）、遮挡纹理（Occlusion Texture）、法线纹理（Normal Texture），一些材质还使用了细节纹理来提供更多的细节表现。

### 18.3.4 线性空间

在使用基于物理的渲染方法渲染整个场景时，我们应该使用**线性空间（Linear Space）**来得到最好的渲染效果。默认情况下，Unity会使用伽马空间（Gamma Space），如果要使用线性空间的话，我们需要在Edit → Project Settings → Player → Other Settings → Color Space中选择Linear选项。图18.17显示了分别在线性空间和伽马空间下场景的渲染结果。





▲图18.17 左边：在线性空间下的渲染结果。右边：在伽马空间下的渲染结果

从图18.17中可以看出，使用线性空间可以得到更加真实的效果。但它的缺点在于，需要一些硬件支持来实现线性计算，但一些移动平台对它的支持并不好。这种情况下，我们往往只能退而求其次，选择伽马空间进行渲染和计算。

那么，线性空间、伽马空间到底是什么意思？为什么线性空间可以得到更加真实的效果呢？这就需要介绍**伽马校正（Gamma Correction）**的相关内容了。实际上，当我们在默认的伽马空间下进行渲染计算时，由于使用了非线性的输入数据，导致很多计算都是在非线性空间下进行的，这意味着我们得到的结果并不符合真实的物理期望。除此之外，由于输出时没有考虑显示器的显示伽马的影响，会导致渲染出来的画面整体偏暗，总是和真实世界不像。

尽管在Unity中我们可以通过之前所说的步骤直接选择在线性空间进行渲染，Unity会在背后为我们照顾好一切，但了解伽马校正的原理对我们理解渲染计算有很大帮助，读者可以在18.4.2节找到更多的解释。

## 18.4 答疑解惑

在上面的内容中，我们首先介绍了PBS实现的数学和理论基础，并简单概括了Unity中Standard Shader的实现原理，以及如何使用它来为不同类型的物体调整适合它们的材质参数。随后，我们通过一个更加复杂的场景，来展示如何在Unity中使用环境光照、实时光源、反射探针以及Standard Shader来渲染一个基于物理渲染的场景。但我们相信，读者在读完后仍有很多困惑，考虑到内容的连贯性，我们未能在文中对某些概念进行展开。在本节中，我们将对一些重要的概念进行更为深入地解释。

### 18.4.1 什么是全局光照

在上面的内容中，我们可以发现全局光照对得到真实的渲染结果有着举足轻重的作用。全局光照，指的就是模拟光线是如何在场景中传播的，它不仅会考虑那些直接光照的结果，还会计算光线被不同的物体表面反射而产生的间接光照。在使用基于物理的着色技术时，当渲染表面上一点时，我们需要计算该点的半球范围内所有会反射到观察方向的入射光线的光照结果，这些入射光线中就包含了直接光照和间接光照。

通常来讲，这些间接光照的计算是非常耗时间的，通常不会用在实时渲染中。一个传统的方法是使用光线追踪，来追踪场景中每一条重要的光线的传播路径。使用光线追踪能得到非常出色的画面效果，因此，被大量应用在电影制作中。但是，这种方法往往需要大量时间才能得到一帧，并不能满足实时的要求。

Unity采用了Enlighten解决方案来让全局光照能够在各种平台上有很好的性能表现。事实上，Enlighten也已经被集成在虚幻引擎（Unreal Engine）中，它已经在很多3A大作中展现了自身强大的渲染能力。总体来讲，Unity使用了实时+预计算的方法来模拟场景中的光照。其中，实时光照用于计算那些直接光源对场景的影响，当物体移动时，光照也会随之发生变化。但正如我们之前所说，实时光照无法模拟光线被多次反射的效果。为了得到更加真实的渲染效果，Unity又引入了预计算光照的方法，使得全局光照甚至在一些高端的移动设备上也可以达到实时的要求。

预计算光照包含了我们常见的光照烘焙，也就是指我们把光源对场景中静态物体的光照效果提前烘焙到一张光照纹理中，然后把这张光照纹理直接贴在这些物体的表面，来得到光照效果。这些光照纹理不仅存储了直接光照的结果，还包含了那些由物体反射得到的间接光照。但是，这些光照纹理无法在游戏运行时不断更新，也就是说，它们是静态的。不过这种方法的确为移动平台的复杂光照模拟提供了一个有效途径。以上提到的这些技术很多读者都已非常熟悉，并可能已经在实际工作中大量使用了它们。

由于静态的光照烘焙无法在光照条件改变时更新物体的光照效果，因此，Unity使用了**预计算实时全局光照（Precomputed Realtime GI）**为我们提供了一个解决途径，来动态地为场景实时更新复杂的光照结果。正如我们之前看到的，使用这种技术我们可以让场景中的物体包含丰富的全局光照效果，例如多次反射等，并且这些计算都是实时的，可以随着光源和物体的移动而发生变化。这是使用之前的实时光照或烘焙光照所无法实现的。

那么，这些是如何实现的呢？它们实际上都利用了一个事实——一旦物体和光源的位置被固定了，这些物体对光线的反弹路径以及漫反射光照（我们假设漫反射光照在各个方向的分布是相同的）也是固定的，也就是说是和摄像机无关的。因此，我们可以使用预计算方法来把这些物体之间的关系提前计算出来，而在实时运行时，只要光源的位置（光源的颜色是可以实时变化的）不变，即便改变了光源颜色和强度、物体材质属性（指的是漫反射和自发光相关的属性），这些信息就一直有效，不需要实时更新。在预计算阶段，**Enlighten**会在由所有静态物体组成的场景上，进行简化的“光线追踪”过程。在这个过程中**Enlighten**会自动把场景分割成很多个子系统，它并不是为了得到精确的光照效果，而是为了得到场景中物体之间的关系。需要注意的是，这些预计算都是在静态物体上进行的，因此，为了利用上述的预计算方法，我们至少需要把场景中的一个物体标识为**Static**（至少需要把**Lightmap Static**勾选上）。一个例外是物体的高光反射，这是和摄像机的位置相关的，**Unity**的解决方案是使用反射探针，正如我们之前看到的那样。对于动态移动的物体来说，我们可以使用光照探针来模拟它的光照环境。因此，在实时运行时，**Unity**会利用预计算得到的信息来计算光照信息，并把它们存储在额外的光照纹理、光照探针或**Cubemap**中，再和物体材质进行必要的光照计算，得到最后的渲染效果。

**Unity**全新的全局光照解决方案可以大大提高一些基于**PC**/游戏机等平台的大型游戏的画面质量，但如果要在移动平台上使用仍需要非常小心它的性能。一些低端手机是不适合使用这种比较复杂的基于物理的渲染，不过，**Unity**会在后续的版本中持续更新和优化。而且随着手机硬件的发展，未来在移动平台上大量使用**PBS**也已经不再是遥不可及

的梦想了。更多关于Unity中全局光照的内容，读者可以在Unity官方手册的**全局光照**（<http://docs.unity3d.com/Manual/GIIntro.html>）一文中找到更多内容，本章最后的扩展阅读部分也会给出更多的学习资料。

### 18.4.2 什么是伽马校正

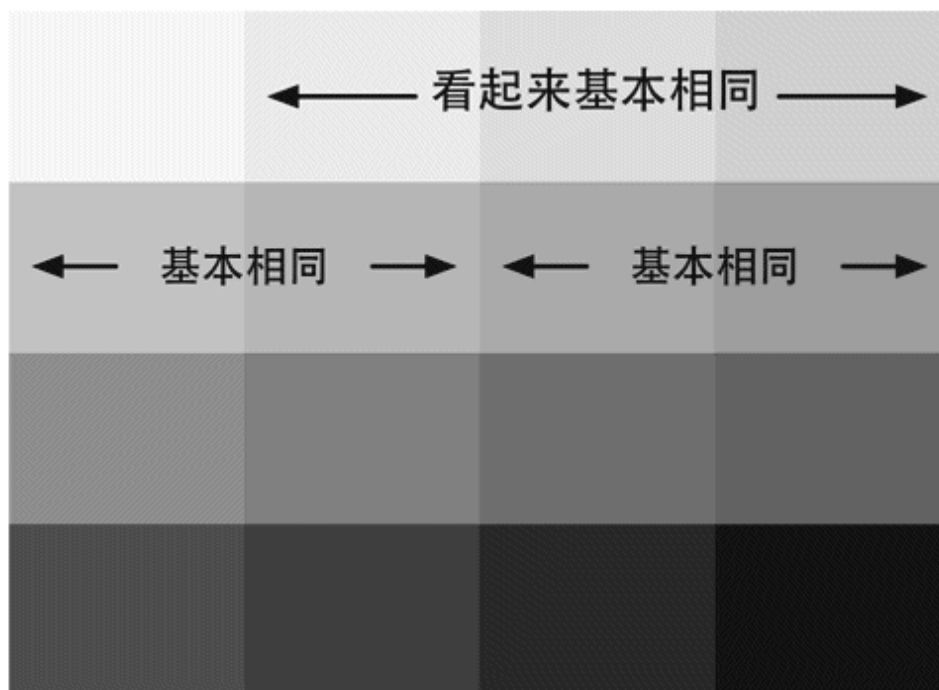
我们在18.3.4节中讲到，要想渲染出更符合真实光照环境的场景就需要使用线性空间。而Unity默认的空间是伽马空间，在伽马空间下进行渲染会导致很多非线性空间下的计算，从而引入了一些误差。而要把伽马空间转换到线性空间，就需要进行**伽马校正（Gamma Correction）**。

相信很多读者都听过伽马校正这个名词，但对于伽马校正是什么、为什么要有它、怎么使用它都存在着很多疑问。伽马校正中的伽马一词来源伽马曲线。通常，伽马曲线的表达式如下：

$$L_{out} = L$$

其中指数部分的发音就是伽马。最开始的时候，人们使用伽马曲线来对拍摄的图像进行**伽马编码（gamma encoding）**。事情的起因可以从在真实环境中拍摄一张图片说起。摄像机的原理可以简化为，把进入到镜头内的光线亮度编码成图像（例如一张JEPG）中的像素。如果采集到的亮度是0，像素就是0亮度是1，像素就是1亮度是0.5，像素就是0.5。如果我们只用8位空间来存储像素的每个通道的话，这意味着0~1区间可以对应256种不同的亮度值。但是，后来人们发现，人眼有一个有趣的特性，就是对光的灵敏度在不同亮度上是不一样的。在正常的光照条件下，人眼对较暗区域的变化更加敏感，如图18.18所示。





▲图18.18 人眼更容易感知暗部区域的变换，而对较亮区域的变化比较不敏感

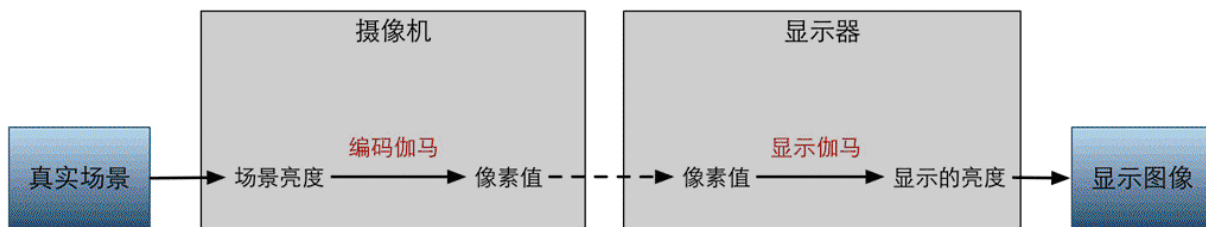
图18.18说明了一件事情，亮度上的线性变化对人眼感知来说是非均匀的。Youtube上有一个名为**Color is Broken**的非常有趣的视频，在这个视频中，作者用了一个非常生动的例子来说明这个现象。当一个屋子的光照由一盏灯增加到两盏灯的时候，人眼对这种亮度变化的感知性要远远大于从101盏灯增加到102盏灯的变化，尽管从物理上来说这两种变化基本是相同的。那么，这和之前讲的拍照有什么关系呢？如果使用8位空间来存储每个通道的话，我们仍然把0.5亮度编码成值为0.5的像素，那么暗部和亮部区域我们都使用了128种颜色来表示，但实际上，对亮部区域使用这么多颜色是种存储浪费。一种更好的方法是，我们应该把更多的空间来存储更多的暗部区域，这样存储空间就可以被充分利用起来了。摄影设备如果使用了8位空间来存储照片的话，会使用大约为0.45的编码伽马来对输入的亮度进行编码，得到一张

编码后的图像。因此，图像中0.5像素值对应的亮度其实并不是0.5，而大约为0.22。这是因为：

$$0.5 \approx 0.22^{0.45}$$

如上所见，对拍摄图像使用的伽马编码使得我们可以充分利用图像的存储空间。但当把图片放到显示器里显示时，我们应该对图像再进行一次解码操作，使得屏幕输出的亮度和捕捉到的亮度是符合线性的。这时，人们发现了一个奇妙的巧合——CRT显示器本身几乎已经自动做了这个解码操作！这又从何说起呢？在早期，CRT（Cathode Ray Tube，阴极射线管）几乎是唯一的显示设备。这类设备的显示机制是，使用一个电压轰击它屏幕上的一种图层，这个图层就可以发亮，我们就可以看到图像了。但CRT显示器有一个特性，它的输入电压和显示出来的亮度关系不是线性的，也就是说，如果我们把输入电压调高两倍，屏幕亮度并没有提高两倍。我们把显示器的这个伽马曲线称为**显示伽马（display gamma）**。非常巧合的是，CRT的显示伽马值大约就是编码伽马的倒数。CRT显示器的这种特性，正好补偿了图像捕捉设备的伽马曲线，人们想，“天呐，太棒了，我们不需要做任何调整就可以让拍摄的图像在电脑上看起来和原来的一样了！”虽然现在CRT设备很少见了，并且后来出现的显示设备有着不同的伽马响应曲线，但是，人们仍在硬件上做了调整来提供兼容性。图18.19展示了编码伽马和显示伽马在图像捕捉和显示时的作用。





▲ 图18.19 编码伽马和显示伽马

随后，微软联合爱普生、惠普提供了sRGB颜色空间标准，推荐显示器的显示伽马值为2.2，并配合0.45的编码伽马就可以保证最后伽马曲线之间可以相互抵消（因为 $2.2 \times 0.45 \approx 1$ ）。绝大多数的摄像机、PC和打印机都使用了上述的sRGB标准。

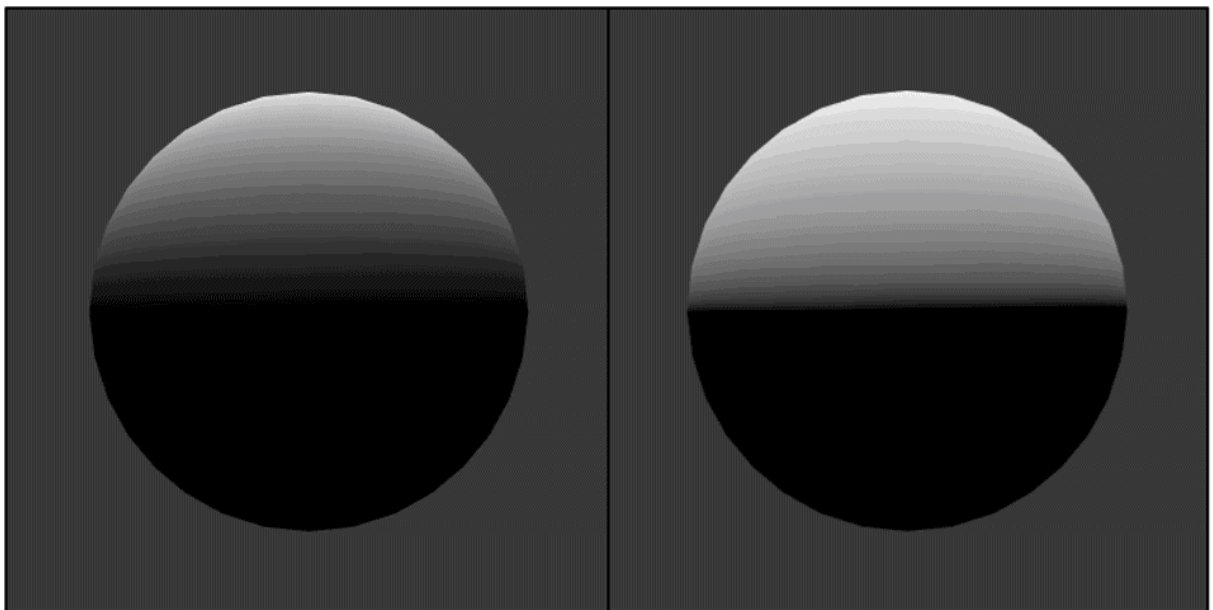
读到现在，读者可能还是有所疑问，这和渲染有什么关系？答案是关系很大。事实上，由于游戏界长期以来都忽视了伽马校正的问题，造成了我们渲染出来的游戏总是暗沉沉的，总是和真实世界不像。由于编码伽马和显示伽马的存在，我们一不小心就可能在非线性空间下进行计算，或是使得输出的图像是非线性的。

对于输出来说，如果我们直接输出渲染结果而不进行任何处理，在经过显示器的显示伽马处理后，会导致图像整体偏暗，出现失真的状况。我们在本书资源的Scene\_18\_4\_2\_a显示了伽马**对光照效果的影响**。在场景Scene\_18\_4\_2\_a中，我们放置了一个球体，并把场景中的环境光照设为全黑，再把平行光的方向设置为从上方直接射到球体表面，球体使用的材质为内置的漫反射材质。图18.20显示了在伽马空间和线性空间下的渲染结果。

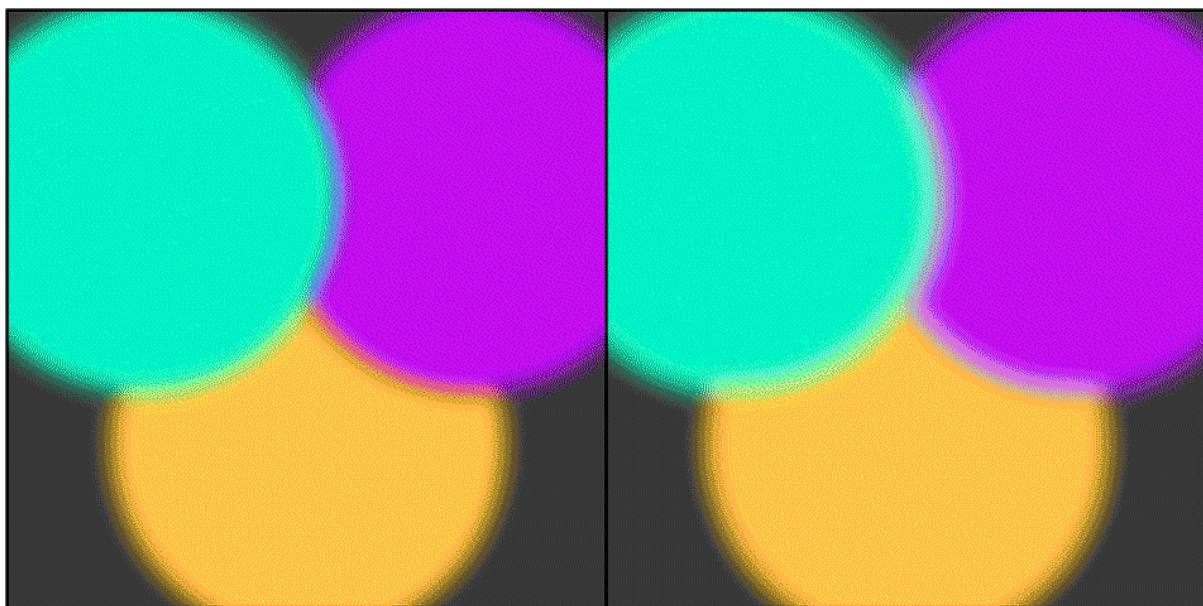
从图18.20可以看出，伽马空间下的渲染结果整体偏暗，一些读者甚至认为这看起来更加正确。然而，实际此时屏幕输出的亮度和球面

的光照结果并不是线性的。假设球面上有一点A，它的法线和光线方向成 $60^\circ$ ，还有一点B，它的法线和光线方向成 $90^\circ$ 。那么，在Shader中计算漫反射光照时，我们会得出A的输出是 $(0.5, 0.5, 0.5)$ ，B的输出是 $(1.0, 1.0, 1.0)$ 。在图18.20的左图中，我们没有进行伽马校正，因此，由于显示器存在显示伽马就引入了非线性关系，也就是说A点的亮度其实并不是B亮度的一半，而约为它的 $1/4$ 。在图18.20的右图中，我们使用了线性空间，Unity会在把像素写入颜色缓冲前进行一次伽马校正，来抵消屏幕的显示伽马的作用，此时得到屏幕亮度才是真正跟像素值成正比的。

伽马的存在还会**对混合造成影响**。在场景Scene\_18\_4\_2\_b中演示了一个简单的场景来说明这个现象。在场景Scene\_18\_4\_2\_b中，我们放置了3个互相重叠的圆，它们使用的材质均为简单的透明混合材质，并使用了一个边界模糊的圆作为输入纹理。场景在伽马空间和线性空间下的效果如图18.21所示。



▲图18.20 左边：伽马空间下的渲染结果，右边：线性空间下的渲染结果



▲图18.21 左边：伽马空间下的混合结果，右边：线性空间下的混合结果

在图18.21左图所示的伽马空间下，我们可以看到在绿色和红色的混合边界处出现了不正常的蓝色渐变。而正确的混合结果应该是如图18.21右边图所示的从绿色到红色的渐变。除此之外，我们也可以看到图18.21左边图中交叉的边界似乎都变暗了。这是因为在混合后进行输出时，显示器的显示伽马导致接缝处颜色变暗。

实际上，渲染中非线性输入最有可能的来源就是纹理。为了充分利用存储空间，大多数图像文件都进行了提前的校正，即已经使用了一个编码伽马对像素值编码。但这意味着它们是非线性的，如果我们在Shader中直接使用纹理采样值就会造成在非线性空间的计算，使得结果和真实世界的结果不一致。我们在使用多级渐远纹理（mipmaps）时 also 需要注意。如果纹理存储在非线性空间中，那么在计算多级渐远纹理时就会在非线性空间里计算。由于多级渐远纹理的计算是种线性计

算——即采样的过程，需要对某个方形区域内的像素取平均值，这样就会得到错误的结果。正确的做法是，我们要把非线性的纹理转换到线性空间后再计算多级渐远纹理。

如上所说，伽马的存在使得我们很容易得到非线性空间下的渲染结果。在游戏渲染中，我们应该保证所有的输入都被转换到了线性空间下，并在线性空间下进行各种光照计算，最后在输出前通过一个编码伽马进行伽马校正后再输出到颜色缓冲中。Unity的颜色空间设置就可以满足我们的需求。当我们选择伽马空间时，实际上就是“放任模式”，不会对Shader的输入进行任何处理，即使输入可能是非线性的；也不会对输出像素进行任何处理，这意味着输出的像素会经过显示器的显示伽马转换后得到非预期的亮度，通常表现为整个场景会比较昏暗。当选择线性空间时，Unity会把输入纹理设置为sRGB模式，在这种模式下，硬件在对纹理进行采样时会自动将其转换到线性空间中；并且，GPU会在Shader写入颜色缓冲前自动进行伽马校正或是保持线性在后面进行伽马校正，这取决于当前的渲染配置。如果我们开启了HDR（见18.4.3节）的话，渲染就会使用一个浮点精度的缓冲。这些缓冲有足够的精度不需要我们进行任何伽马校正，此时所有的混合和屏幕后处理都是在线性空间下进行的。当渲染完成要写入显示设备的后备缓冲区（back buffer）时，再进行一次最后的伽马校正。如果我们没有使用HDR，那么Unity就会把缓冲设置成sRGB格式，这种格式的缓冲就像一个普通的纹理一样，在写入缓冲前需要进行伽马校正，在读取缓冲时需要再进行一次解码操作。如果此时开启了混合（像我们之前的那样），在每次混合时，硬件会首先把之前颜色缓冲中存储的颜色值转换回线性空间中，然后再与当前的颜色进行混合，完成后再进行伽马

校正，最后把校正后的混合结果写入颜色缓冲中。这里需要注意，透明通道是不会参与伽马校正的。

然而，Unity的线性空间并不是所有平台都支持的，例如，移动平台就无法使用线性空间。此时，我们就需要自己在Shader中进行伽马校正。对非线性输入纹理的校正代码通常如下：

```
float3 diffuseCol = pow(tex2D( diffTex, texCoord ), 2.2 );
```

在最后输出前，对输出像素值的校正代码通常如下面这样：

```
fragColor.rgb = pow(fragColor.rgb, 1.0/2.2);  
return fragColor;
```

但是，手工对输出像素进行伽马校正会在使用混合时出现问题。这是因为，校正会导致写入颜色缓冲内的颜色是非线性的，这样混合就发生在非线性空间中。一种解决方法是，在中间计算时不要对输出颜色值进行伽马校正，但在最后需要进行一个屏幕后处理操作来对最后的输出进行伽马校正，也就是说我们需要保证伽马校正发生在渲染的最后一步中，但这可能会造成一定的性能损耗。

你会说，伽马这么麻烦，什么时候可以舍弃它呢？如果有一天我们对图像的存储空间能够大大提升，通用的格式不再是8位时，例如是32位时，伽马也许就会消失。因为，我们有足够多的颜色空间可以利用，不需要为了充分利用存储空间进行伽马编码的工作了。这就是我们下面要讲的HDR。

### 18.4.3 什么是HDR



在使用基于物理的渲染时，我们经常会听到一个名词就是**HDR**。**HDR**是**High Dynamic Range**的缩写，即高动态范围，与之相对的是**低动态范围（Low Dynamic Range, LDR）**。那么这个动态范围是指什么呢？通俗来讲，动态范围指的就是最高的和最低的亮度值之间的比值。在真实世界中，一个场景中最亮和最暗区域的范围可以非常大，例如，太阳发出的光可能要比场景中某个影子上的点的亮度要高出几万倍，这些范围远远超过图像或显示器能够显示的范围。通常在显示设备使用的颜色缓冲中每个通道的精度为8位，意味着我们只能用这256种不同的亮度来表示真实世界中所有的亮度，因此，在这个过程中一定会存在一定的精度损失。早期的拍摄设备利用人眼的特点，使用了伽马曲线来对捕捉到的图像进行编码，尽可能充分地利用这些有限的存储空间，这点我们已经在18.4.2节解释过了。然而，**HDR**的出现给我们带来了新的希望，**HDR**使用远远高于8位的精度（如32位）来记录亮度信息，使得我们可以表示超过0~1内的亮度值，从而可以更加精确地反映真实的光照环境。尽管我们最后还是需要把信息转换到显示设备使用的**LDR**内，但中间的计算却可以让我们得到更加真实可信的效果。**Nvidia**曾总结过使用**HDR**进行渲染的**动机**：让亮的物体可以真的非常亮，暗的物体可以真的非常暗，同时又可以看到两者之间的细节。

使用**HDR**来存储的图像被称为高动态范围图像（**HDRI**），例如，我们在18.3节中就是使用了一张**HDRI**图像来作为场景的**Skybox**。这样的**Skybox**可以更加真实地反映物体周围的环境，从而得到更加真实的反射效果。不仅如此，**HDR**对与光照叠加也有非常重要的作用。如果我们的场景中有很多光源或是光源强度很大，那么一个物体在经过多次光照渲染叠加后最终得到的光照亮度很可能会超过1。如果没有使用

HDR，这些超过1的部分全部会截取到1，使得场景丢失了很多亮部区域的细节。但如果开启了HDR，我们就可以保留这些超过范围的光照结果，尽管最后我们仍然需要把它们转换到LDR进行显示，但我们可以使用**色调映射（tonemapping）**技术来控制这个转换的过程，从而允许我们最大限度地保留需要的亮度细节。

HDR的使用可以允许我们在屏幕后处理中拥有更多的控制权。例如，我们常常同时使用HDR和Bloom效果。我们曾在12.5节解释了Bloom特效的实现原理，Bloom效果需要检测屏幕中亮度大于某个阈值的像素，把它们提取出来后进行模糊，再叠加到原图像中。但是，如果不使用HDR的话，我们只能使用小于1的阈值来提取需要的像素，但很多时候我们实际上是需要提取那些非常亮的区域，例如车窗上对太阳的强烈反光。由于没有使用HDR，这些值实际上很可能和街上一些颜色偏白的区域几乎一样，造成不希望的区域也会出现泛光的效果。如果我们使用HDR，这些就都可以解决了，我们只需要使用超过1的阈值来只提取那些非常亮的区域即可。

总体来说，使用HDR可以让我们不会丢失高亮度区域的颜色值，提供了更真实的光照效果，并为一些屏幕后处理提供了更多的控制能力。但HDR也有自身的缺点，首先由于使用了浮点缓冲来存储高精度图像，不仅需要更大的显存空间，渲染速度会变慢，除此之外，一些硬件并不支持HDR。而且一旦使用了HDR，我们无法再利用硬件的抗锯齿功能。事实上，在Unity中如果我们同时打开了硬件的抗锯齿（在Edit → Project Settings → Quality → Anti Aliasing中打开）和摄像机的HDR，Unity会发出警告来提示我们由于开启了抗锯齿，因此，无法使



用HDR缓冲。尽管如此，我们可以使用基于屏幕后处理的抗锯齿操作来弥补这一点。

在Unity中使用HDR也非常简单，我们可以在Camera组件面板中打开HDR选项即可。此时，场景就会被渲染到一个HDR的图像缓冲中，这个缓冲的精度范围可以远远超过0~1。最后，我们可以再使用一个色调映射的屏幕后处理脚本来把HDR图像转换到LDR图像进行显示。读者可以在Unity官方手册中的**高动态范围渲染**一节

(<http://docs.unity3d.com/Manual/HDR.html>) 以及本章最后的扩展阅读中找到更多的内容。

#### 18.4.4 那么，PBS适合什么样的游戏

在把PBS引入当前的游戏项目之前，我们需要权衡一下它的优缺点。需要再次提醒读者的是，PBS并不意味着游戏画面需要追求和照片一样真实的效果。事实上，很多游戏都不需要刻意去追求与照片一样的真实感，玩家眼中的真实感大多也并不是如此。PBS的优点在于，我们只需要一个万能的Shader就可以渲染相当一大部分类型的材质，而不是使用传统的做法为每种材质写一个特定的Shader。同时，PBS可以保证在各种光照条件下，材质都可以自然地 and 光源进行交互，而不需要我们反复地调整材质参数。

然而，在使用PBS时我们也需要考虑到它带来的代价。如上面提到的，PBS往往需要更复杂的光照配合，例如大量使用光照探针和反射探针等。而且PBS也需要开启HDR以及一些必不可少的屏幕特效，例如抗锯齿、Bloom和色调映射，如果这些屏幕特效对当前游戏来说需要消耗过多的性能，那么PBS就不适合当前的游戏，我们应该使用传统的

Shader来渲染游戏。使用PBS对美工人员来说同样是个挑战。美术资源的制作过程和使用传统的Shader有很大不同，普通的法线纹理+高光反射纹理的组合不再适用，我们需要创建更细腻复杂的纹理集，包括金属值纹理、高光反射纹理、粗糙度纹理、遮挡纹理，有些还需要使用额外的细节纹理来给材质添加更多的细节表面。除了使用图片扫描的传统辅助方法外，这些纹理的制作通常还需要更专业的工具来绘制，例如**Allegorithmic Substance Painter**和**Quixel Suite**。

## 18.5 扩展阅读

Unity官方提供了很多学习PBS的资料。在**Unity**官方博客中的**全局光照**一文（[global-illumination-in-unity-5](#)）中，简明地阐述了全局光照的解决方案。在另外两篇博客（[working-with-physically-based-Shading-a-practical-approach/](#)、[physically-based-shading-in-unity-5-a-primer/](#)）中，介绍了Standard Shader的用法和注意事项。官方项目也是很好的学习资料。Unity开放了基于物理着色器的示例项目**Viking Village**以及两个更小的示例项目**Shader Calibration Scene**和**Corridor Lighting Example**来着重介绍如何使用Unity 5全新的Standard Shader和全局光照系统。看过Unity 5宣传视频的读者想必对Unity 5制作出来的电影短片**The Blacksmith**印象深刻，尽管Unity没有开放出完整的工程，但把许多关键的技术实现放到了资源商店里，例如，人物角色使用的Shader、头发使用的Shader、人物阴影、大气次散射等，这些都是非常好的学习资料。除此之外，Unity还提供了一些相关教程供新手学习，读者可以在**图形**的教程板块（<http://unity3d.com/cn/learn/tutorials/topics/graphics>）下找到很多相关教程。例如，在**Unity 5的光照概览**

(<http://unity3d.com/cn/learn/tutorials/modules/beginner/unity-5/unity5-lighting-overview>) 中, 介绍了Unity 5中使用的各种全局光照技术; **光照和渲染** (<https://unity3d.com/cn/learn/tutorials/modules/beginner/graphics/lighting-and-rendering>) 一文更加详细地介绍了Unity 5中各种光照的实现细节, 以及一些设置场景光照时的注意事项; 在**Standard Shader**的视频教程 (<http://unity3d.com/cn/learn/tutorials/modules/beginner/5-tutorials/standard-shader>) 中, Unity介绍了Standard Shader的基本用法以及和光照之间的配合。

近年来, Unity官方在Unite、SIGGRAPH等大会上也分享不少关于PBS的技术资料。在Unite 2014会议上, Anton Hand在他的演讲中给出了很多关于如何创建PBS中使用的资源的最佳实践; Renaldas Zioma和Erland Körner讲解了如何在Unity 5中更加有效地使用PBS。在SIGGRAPH 2015会议上, 来自Unity的技术人员分享了**The Blacksmith**的环境制作过程。

如果读者希望更深入地学习PBS的理论和实践, 可以在近年来的SIGGRAPH课程上找到非常丰富的资料。SIGGRAPH自2006年起开始出现与PBS相关的课程, 更是连续4年(2012~2015)由来自各大游戏公司和影视公司的技术人员分享他们在PBS上的实践。例如在2012年的课程上, Disney公布了他们在离线渲染时使用的BRDF模型, 这也是Unity等很多游戏引擎使用的PBR的理论基础。Kostas Anagnostou在他的文章中列出了非常多的关于PBR的相关文章, 包括我们上面提到的SIGGRAPH课程, 强烈建议有兴趣的读者去浏览一番。

国内的相关资料则相对较少。龚敏敏在他的KlayGE引擎中引入了PBS，并写了系列博文来简明地阐述其中的理论基础。在知乎专栏 **Behind the Pixels** (<http://zhuanlan.zhihu.com/graphics>) 中，作者给出了3篇关于基于物理着色的系列文章。

## 18.6 参考文献

[1] Hoffman N. Background: physics and math of shading[C]//Fourth International Conference and Exhibition on Computer Graphics and Interactive Techniques, Anaheim, USA. 2013: 21-25。

[2] Burley B, Studios W D A. Physically-based shading at disney[C]//ACM SIGGRAPH. 2012: 1-7。

[3] Walter B, Marschner S R, Li H, et al. Microfacet models for refraction through rough surfaces[C]//Proceedings of the 18th Eurographics conference on Rendering Techniques. Eurographics Association, 2007: 195-206。

[4] Beckmann P, Spizzichino A. The scattering of electromagnetic waves from rough surfaces[J]. Norwood, MA, Artech House, Inc., 1987, 511 p., 1987, 1。

[5] Torrance K E, Sparrow E M. Theory for off-specular reflection from roughened surfaces[J]. JOSA, 1967, 57(9): 1105-1112。

[6] Smith B G. Geometrical shadowing of a random rough surface[J].  
Antennas and Propagation, IEEE Transactions on, 1967, 15(5): 668-671 °

[7] Blinn J F. Models of light reflection for computer synthesized  
pictures[C]//ACM SIGGRAPH Computer Graphics. ACM, 1977, 11(2):  
192-198 °

[8] Schlick C. An inexpensive BRDF model for physically-based  
rendering[C]//Computer graphics forum. 1994, 13(3): 233-246 °

## 第19章 Unity 5更新了什么

Unity 5相较于之前的版本来说，在Shader方面做了许多重要的更新。一些更新很容易被大家察觉，例如，如果读者直接把在Unity 4中使用的一些Shader源代码粘贴到Unity 5中，往往会发现和Unity 4中得到的渲染结果不尽相同，甚至还会报错。本章将会对Unity 5进行的一些重要更新（仅关注Shader方面的更新）进行解释，来帮助读者加深对Unity Shader的理解。

### 19.1 场景“更亮了”

如果你曾经学习或阅读过Unity 5之前的一些Shader源码的话，往往会在计算漫反射时发现类似下面的代码：

```
// Unity 5之前的shader经常包含了类似下面的代码，  
// 而在Unity 5中，我们不需要再进行x2的操作  
c.rgb = s.Albedo * _LightColor0.rgb * (diff * atten * 2);
```

这类代码通常会在光照结果的最后乘以系数2，而作者往往解释说，因为不乘以2的话场景会看起来很暗。但是，如果我们仍然在Unity 5中编写类似上面的代码，场景就会看起来变亮了，这通常不是我们希望看到的。Unity 5之前的Shader中需要乘以2是一个历史遗留原因，并最终在Unity 5中得到了修正。在Unity 5中，光照的强度被自动增强到原来的两倍。这意味着，如果我们在场景中放置一个纯白色的平面，同时让一个平行光从它的正上方垂直照射到它的表面，那么平

面得到的漫反射光照结果就是平行光本身的颜色。而在Unity 5之前的版本中，上述的平面并不会得到和光源颜色一致的结果。

因此，如果读者直接从之前的项目中使用现成的Shader代码并把它移植到Unity 5中，需要去掉光照计算中乘以2的部分，来得到和之前一致的光照结果。

## 19.2 表面着色器更容易“报错了”

如果读者把一些老版本下使用的表面着色器代码直接粘贴到Unity 5中使用，可能会发现原本并没有报错的代码在Unity 5下报错了，这些报错信息通常是指Shader中的数学指令或插值寄存器的数目超过了限制，并提示需要使用更高的Shader Model，如SM 3.0。

这些报错信息的出现，是因为Unity 5的表面着色器在背后进行了更多的计算。我们在第17章中解释过表面着色器的实现原理，概括来说就是Unity会在背后把表面着色器转换成对应的顶点/片元着色器。这些转换过程通常是有规律可循的，而在Unity 5中，Unity在转换过程中使用了更多的计算和插值寄存器，从而造成一些自定义的表面着色器可能会在新版本中报错。这些新添加的计算和插值寄存器通常是为了计算阴影、雾效、非统一缩放模型的法线变换矩阵。在Unity 5之前，如果需要使用法线纹理，Unity会把观察方向和光照方向在顶点着色器中变换到切线空间下再传递给片元着色器，但Unity 5则选择首先在顶点着色器中计算从切线空间到世界空间的变换矩阵，再在片元着色器中把法线变换到世界空间下，从而需要使用更多的插值寄存器来存储变换矩阵。由于Unity默认的Shader Model版本为2.0，这些新添加



的计算再加上一些自定义的变量和计算就很有可能会超过SM 2.0中对计算指令和插值寄存器数目的限制。

对上述问题的解决方法也很简单。一种方法是直接使用更高的Shader Model，例如，在Shader中添加如下代码来指明使用SM 3.0:

```
#pragma target 3.0
```

另一个方法是减少表面着色器背后的计算，这可以通过表面着色器的编译指令来实现。例如，我们可以通过类似下面的编译指令，来指明不需要为该物体计算阴影纹理坐标（不接收阴影）、光照纹理坐标以及雾效:

```
#pragma surface surfaceFunction lightModel noshadow nolightmap  
nofog
```

## 19.3 当家做主：自己控制非统一缩放的网格

Unity 5的另一个重要的改进是，非统一缩放的网格不再由Unity提前在CPU中处理了。我们曾在4.7节讲到过非统一缩放对法线变换的影响，非统一缩放的网格需要使用原变换矩阵的逆转置矩阵来变换法线才可以得到正确的变换结果。然而，在Unity 5之前的版本中，我们并不需要在Shader中考虑非统一缩放带来的种种影响，因为传到Shader中的数据已经不存在非统一缩放了。那么，这是如何做到的呢？Unity 5之前的版本会在CPU中把涉及非统一缩放的模型变换成统一缩放的模型，也就是说，Unity会在CPU中再创建一个和非统一缩放模型空间大小相同，但只包含统一缩放的模型。因此，我们常常会在一些较旧的Shader版本中看到类似下面的代码:

```
// #define SCALED_NORMAL (v.normal * unity_Scale.w)
float3 worldNormal = mul((float3x3)_Object2World, SCALED_NORMAL);
```

上面的代码把法线从模型空间变换到世界空间下，由于只包含统一缩放，因此，代码首先使用统一缩放系数`unity_Scale.w`（在Unity 4.x中，该值表示的值为1/统一缩放系数）来得到归一化后的法线，然后再使用模型空间到世界空间的变换矩阵直接变换法线方向。Unity 5之前采用的这种做法的好处是，我们不需要在渲染中考虑非统一缩放的影响，而它的缺点是，CPU的计算消耗会更大，而且需要占用更多内存空间来存储这些重新缩放的模型。

Unity 5正式抛弃了之前的做法，它直接将原顶点信息和包含非统一缩放的矩阵传递给Shader，因此，`unity_Scale`也就没有意义了。如果我们需要在顶点/片元着色器中变换顶点法线，就需要时刻小心非统一缩放的影响，以及需要对变换后的法线进行手动归一化的操作。

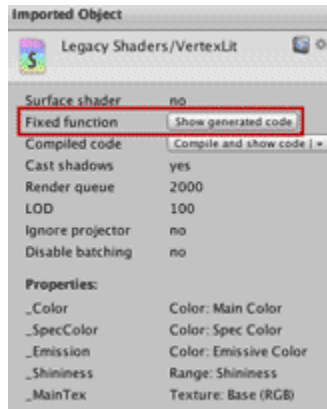
## 19.4 固定管线着色器逐渐退出舞台

我们在3.4.3节中讲到，Unity支持的着色器形式包含了固定管线的着色器类型。固定管线着色器是在可编程着色器出现之前，GPU大量使用的着色器形式。它的工作方式就像是一个包含了很多开关和配置的黑箱子，我们可以通过开启或关闭某些功能来让GPU进行相应的渲染。在Unity最开始出现的时期里（2005年6月，Unity 1.0.1发布），使用固定管线的GPU还占据了一定的市场份额，因此，Unity从那时开始就始终支持编写固定管线的着色器。

然而，实际上很多平台早已不支持固定管线着色器。例如，OpenGL 1.5是最后一个使用固定管线编程的OpenGL版本，从OpenGL 2.0（2004年发布）开始，就只支持可编程管线的着色器。Unity仍然保留对固定管线着色器的支持，是出于两个主要原因：首先，有很多项目和资源包都大量使用了固定管线着色器，其次是固定管线着色器的代码通常要比实现相同功能的顶点/片元着色器少很多。现在Unity支持的绝大多数平台实际已经完全不再支持固定管线编程，Unity需要在背后把我们编写的固定管线着色器转换成相应的可编程管线着色器。

但是，这样的做法有很多弊端。首先，诸如PS4和XboxOne这样的平台并不支持Unity的固定管线着色器（这点在Unity 5.2中得到了改善），这主要是因为想要在这些平台上实时生成着色器代码是很困难的。其次，虽然固定管线着色器代码比较简单，但这些着色器能够实现的效果非常有限，最后，我们往往仍然需要使用灵活性更高的顶点/片元着色器来替代。

Unity 5.x版本对固定管线着色器的导入和编译进行了优化。截止到Unity 5.2版本，所有固定管线着色器都会在导入时被转换成真正的顶点/片元着色器，并且已经支持所有平台，包括游戏机平台。我们还可以在Shader的导入面板中查看固定管线着色器生成的顶点/片元着色器，如图19.1所示。



▲ 图19.1 在shader的导入面板中，单击图中按钮可查看Unity 为该固定管线着色器生成的顶点/片元着色器代码

但缺点是，以前一些固定管线着色器的功能，例如，使用TexGen命令来生成纹理坐标以及进行纹理坐标变换的矩阵操作等，已经被抛弃了。而且，我们也不可以再在脚本中使用类似new Material(“fixed function shader string”)的代码来实时创建一个固定管线的着色器。除此之外，我们也不可以再混用可编程和固定管线的着色器。

实际上，由于Unity目前支持的所有平台都已经抛弃了固定管线着色器，我们已经没有必要再使用固定管线着色器来进行渲染了。如果读者希望了解更多Unity 5对固定管线的优化，可以参见Unity图形工程师Aras的博客。

## 第20章 还有更多内容吗

我们相信一本几十万字的书籍并不能满足一些读者对于渲染强烈的求知欲。在本书的最后，我们会给出许多优秀的学习资料来帮助读者进行下一步的学习。

### 20.1 如果你想深入了解渲染的话

Unity Shader实际是建立在OpenGL、DirectX这样更加基础的图像编程接口上的。这样的封装可以为我们节省很多工作，但可能会影响我们对底层工作方式的理解。这些图像编程接口都有各自非常出色的学习资料，例如OpenGL有非常有名的红宝书《OpenGL编程指南》<sup>[1]</sup>和蓝宝书《OpenGL超级宝典》<sup>[2]</sup>。更多的参考书可以在叶劲峰（网名：Milo Yip）的豆列**计算机图形：入门/API类**（<http://www.douban.com/doulist/1445744/>）中找到。

GPU精粹系列书籍<sup>[3][4][5]</sup>中包含许多游戏和其他实时渲染中使用的高级渲染技术。与之类似的还有GPU Pro系列书籍<sup>[6]</sup>和ShaderX系列书籍<sup>[7]</sup>。这些内容相对比较高深，大都来源于行业内的精英对各种渲染技术的总结，希望深入了解渲染各个方面的读者一定不可以错过。叶劲峰在他的豆列**计算机图形：Gems类**（<http://www.douban.com/doulist/1445745/>）中总结了更多的图形学精粹系列书籍。

尽管本书关注的是游戏中使用的实时渲染技术，但一些基于光线追踪等方式的渲染方法同样是图形学中的重点。在《**Physically based rendering: From theory to implementation**》<sup>[8]</sup>一书中，作者介绍并实现了基于物理渲染的框架，这是学习光线追踪和PBS的非常好的资料。

最后，我们不得不提起被誉为图形程序员专著的《**Real-time Rendering, third Edition**》<sup>[9]</sup>一书。在该书出版时，几乎涵盖了实时渲染中的所有相关技术，作者在书中给出了大量的参考文献，并在网上维护了一个专门的页面来总结实时渲染中使用的各个技术和资料。

在学术方面，图形学相关的会议和论坛是开阔视野、学习前沿渲染技术的绝佳途径。**SIGGRAPH**会议是图形学领域最顶级的会议，每年来自世界各地的顶尖学者和行业精英都会汇聚一堂，展示这一年中他们在图形学领域的工作和进展。与之类似的会议还有，**SIGGRAPH Asia**、**Eurographics**、**Symposium on Interactive 3D Graphics and Games**等会议，读者可以在**Ke-Sen Huang**的主页中找到历年在这些会议上发表的论文。需要特别提出的是，每年**SIGGRAPH**上的**SIGGRAPH Course**中都会有很多来自游戏行业的技术人员分享他们在游戏图像方面的进展，除了在第18章中提到的课程**Physically Based Shading in Theory and Practice**外，**Advances in Real-Time Rendering**系列课程同样是非常出色的学习资料。在这个课程中，来自艺电、育碧、Epic等知名游戏公司的技术人员将阐述他们是如何在游戏中使用各种复杂的渲染技术来实现次世代游戏画面的。自2006年起，该课程已经在**SIGGRAPH Course**上连续举办了十届。另一个与游戏息息相关的会议是游戏开发者会议（**Game Developers Conference, GDC**），每年的

GDC会议都会汇集全世界的游戏开发者。自2009年，中国也迎来了GDC China，给中国的游戏开发者提供了更多的行业交流机会。

除了上述提到的书籍和会议外，一些非常有趣的网站也可以帮助开阔我们的视野。在Shadertoy网站上，你可以看到来自全世界的人们是如何只用一个片元着色器来实现各种或恢弘壮丽、或经典怀旧的场景的。与之类似的还有GLSL Sandbox Gallery网站。我们相信，在浏览了这些网站后，你会再一次被Shader能实现的效果所震撼。

## 20.2 世界那么大

我们曾听到很多声音，抱怨Unity Shader学习资料甚少，尽管我们希望通过这本书来改善这样的情况，但不可否认的是，仅靠一本书恐怕无法让一个人从技术“小白”成长为行业大牛。对于渲染这样牵扯到很多复杂知识的领域来说，一本书更是无法详细地解释这其中的方方面面。实际上，网络上有许多关于这方面的英文资料，我们能够体会许多英语能力欠佳的开发者在这方面的苦恼，但如果你永远不阅读英文资料，那么你将错过一大片“森林”。尽管有不少英文资料不断被引进国内，并有了中译版本，但是由于翻译质量问题等因素给初学者带来了不少的阅读障碍。更何况，还有数之不尽的优秀的英文资料是仍然没有被引入的。

事实上，很多英文资料中使用的英语大多是基础英语，在一些翻译软件的帮助下，阅读并理解这些内容并没有想象中的那么困难。在作者身边也有不少对学习英语十分苦恼的朋友，在经过一段时间的坚



持后，他们普遍反映阅读英文书籍越来越轻松。世界那么大，不要让语言成为阻碍你前进的绊脚石。

最后，我们真心地希望，本书可以为你的Shader学习之旅打开一扇大门，让你离制作心目中优秀游戏的心愿更近一步。若是如此，那想必就是我们最大的欣慰了。

## 20.3 参考文献

[1] Shreiner D, Sellers G, Kessenich J M, et al. OpenGL programming guide: The Official guide to learning OpenGL, version 4.3[M]. Addison-Wesley, 2013. 中译本：《OpenGL编程指南（第8版）》。

[2] Wright R S, Haemel N, Sellers G M, et al. OpenGL SuperBible: comprehensive tutorial and reference[M]. Pearson Education, 2010. 中译本：《OpenGL超级宝典（第5版）》。

[3] Fernando R, Haines E, Sweeney T. GPU gems: programming techniques, tips, and tricks for real-time graphics[J]. Dimensions, 2001, 7(4): 816. 中译本：《GPU精粹1》。

[4] Pharr M, Fernando R. Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation[M]. Addison-Wesley Professional, 2005. 中译本：《GPU精粹2》。

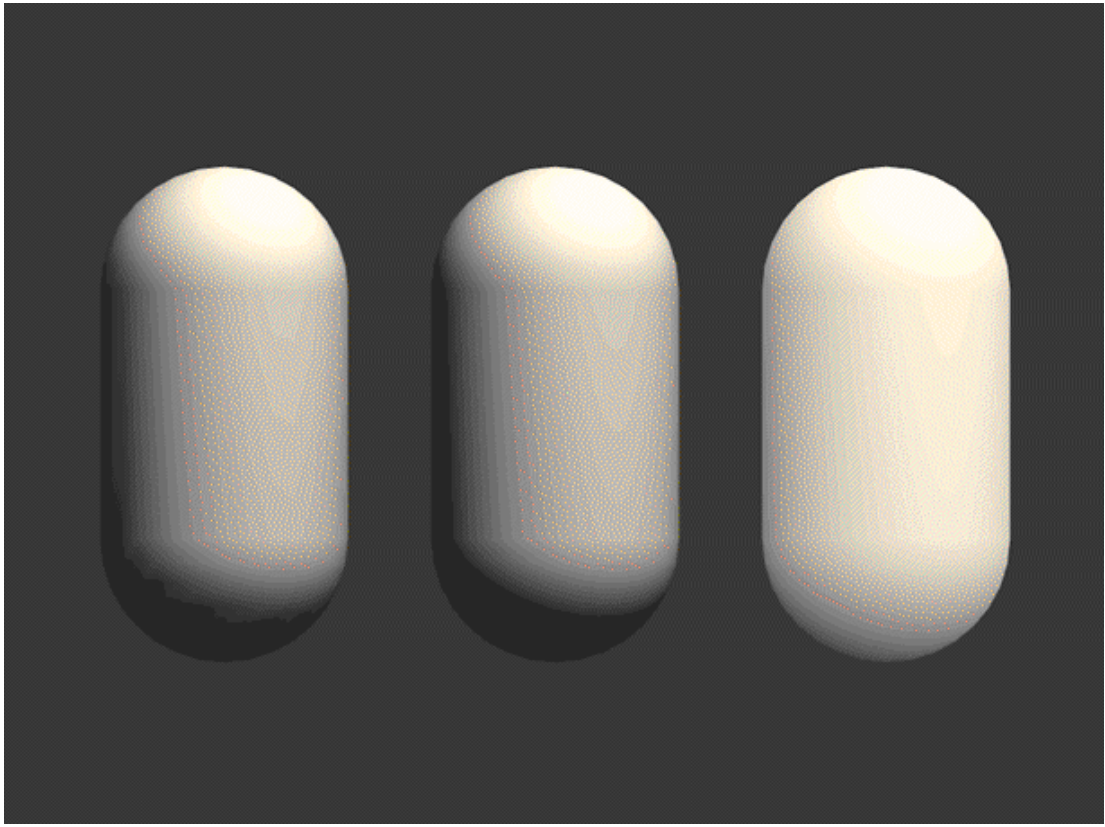
[5] Nguyen H. Gpu gems 3[M]. Addison-Wesley Professional, 2007. 中译本：《GPU精粹3》。

[6] GPU Pro 5: Advanced Rendering Techniques[M]. CRC Press, 2014 °

[7] Engel W. ShaderX7: Advanced Rendering with DirectX and OpenGL (Shaderx Series)[M]. Charles River Media, Inc., 2009 °

[8] Pharr M, Humphreys G. Physically based rendering: From theory to implementation[M]. Morgan Kaufmann, 2010 °

[9] Akenine-Möller T, Haines E, Hoffman N. Real-time rendering[M]. CRC Press, 2008 °



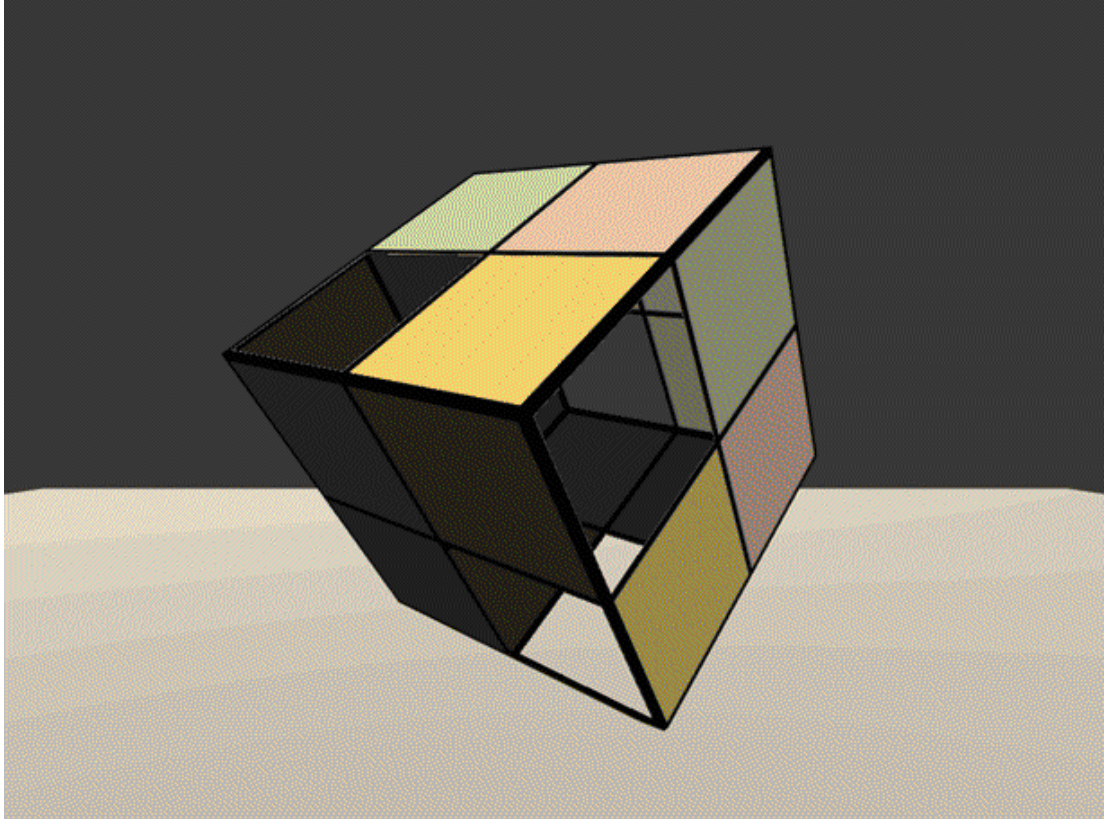
▲ 6.4节：逐顶点漫反射光照、逐像素漫反射光照和半兰伯特光照



▲ 7.2节：使用法线纹理

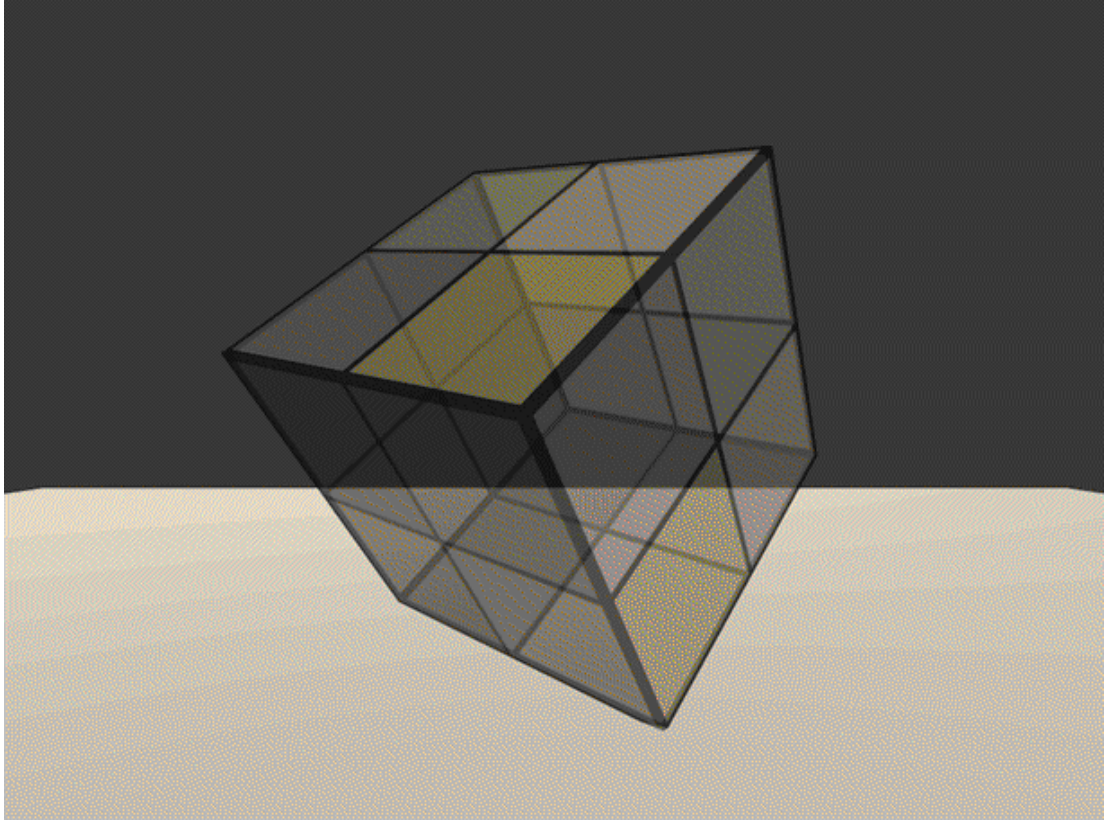


▲ 7.3节：使用渐变纹理来控制漫反射光照



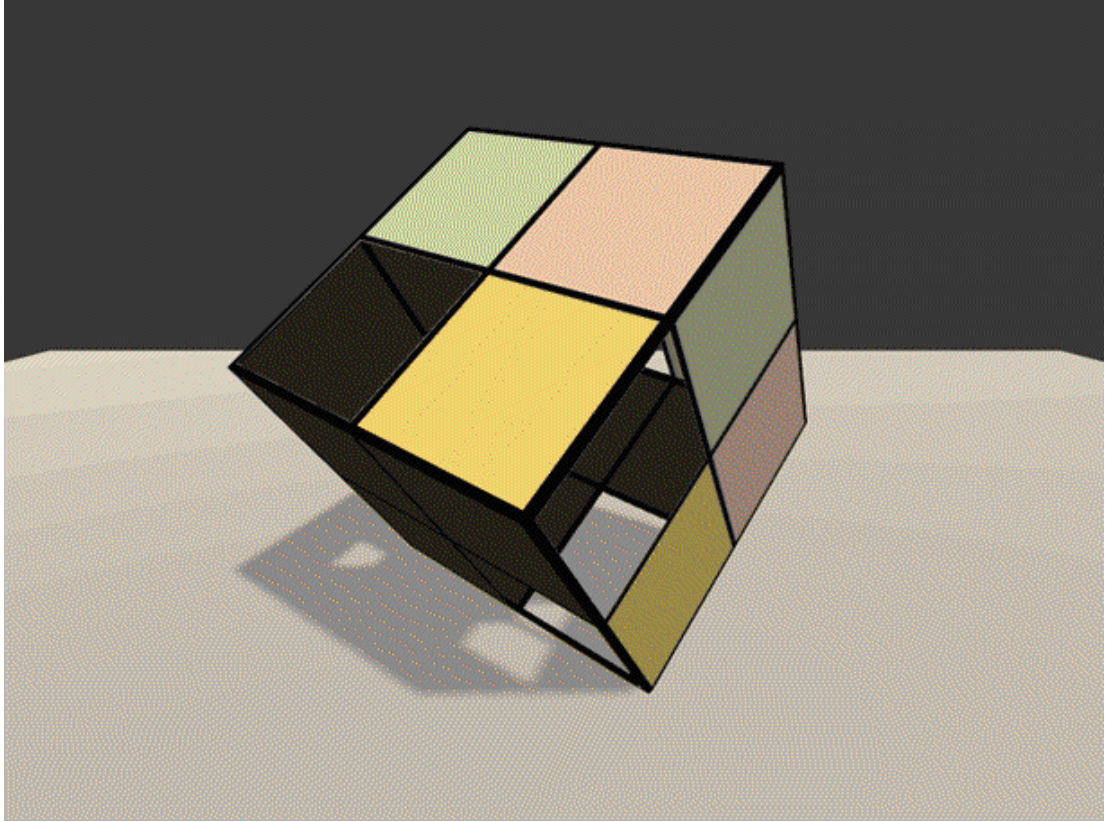
▲8.7.1节：透明度测试的双面渲染效果



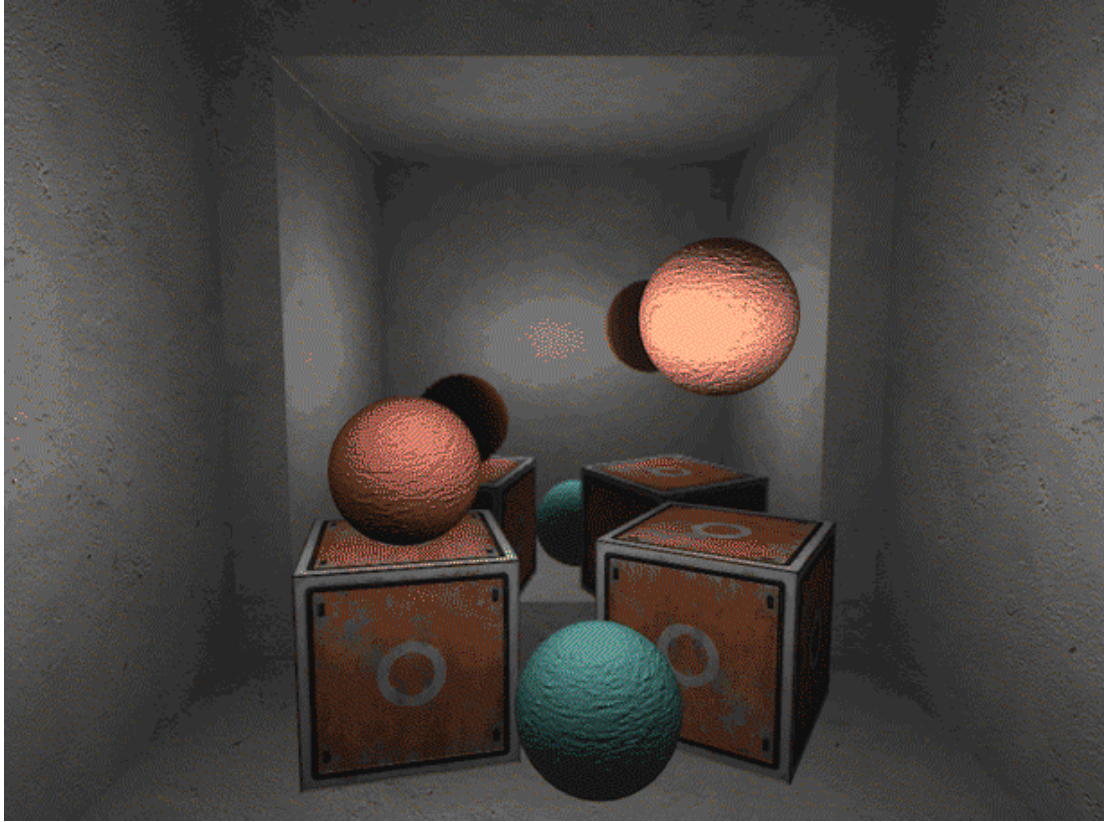


▲8.7.2节：透明度混合的双面渲染效果

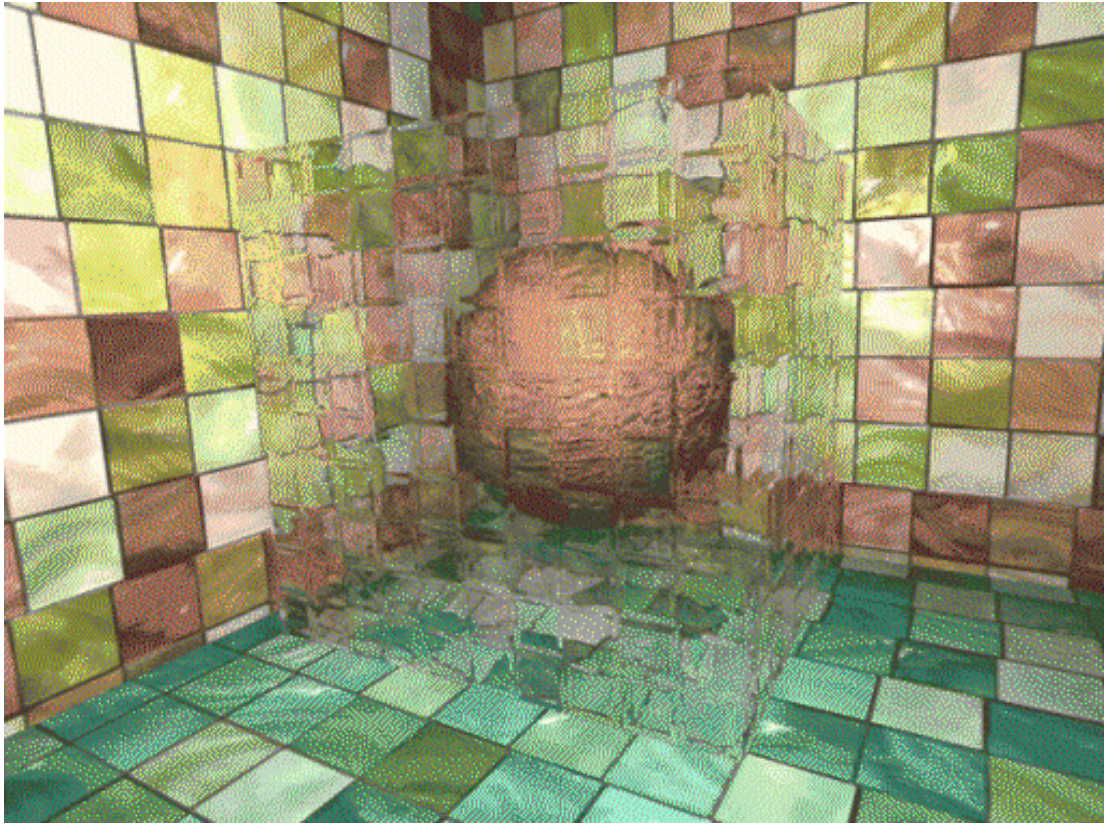




▲ 9.4节：透明度测试的正确阴影效果

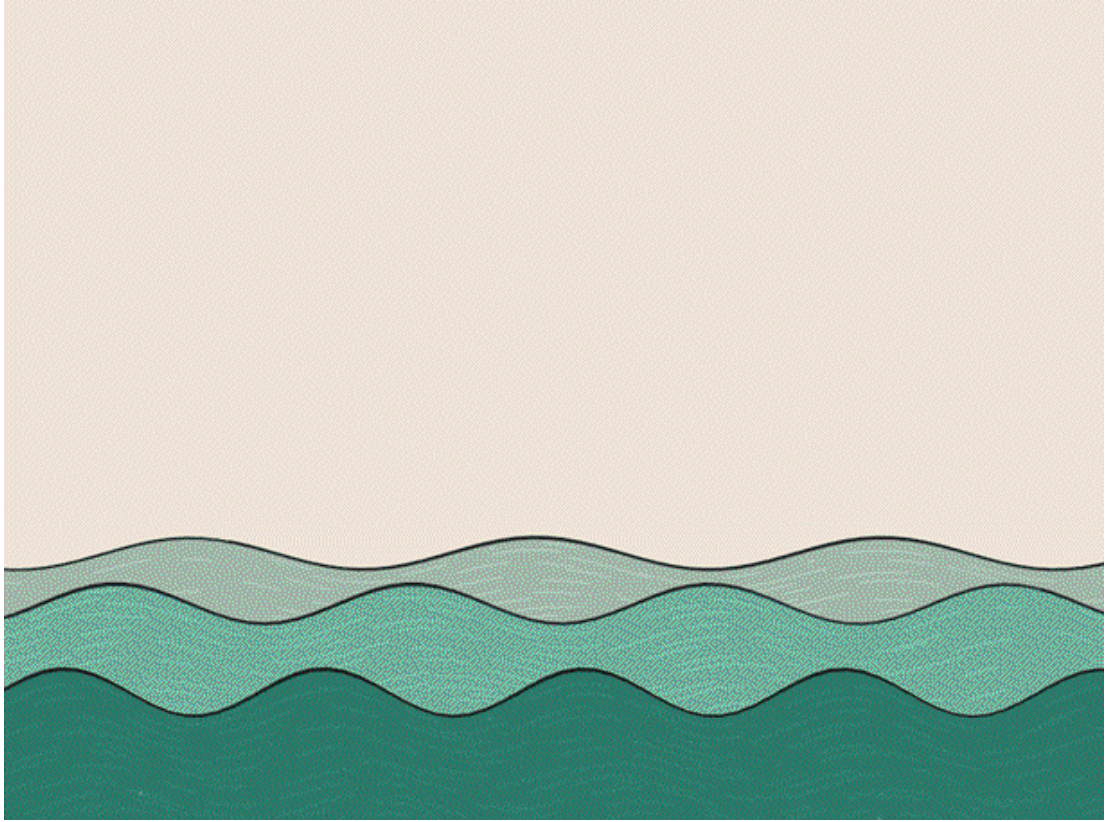


▲ 10.2.1节：使用渲染纹理来实现镜子效果

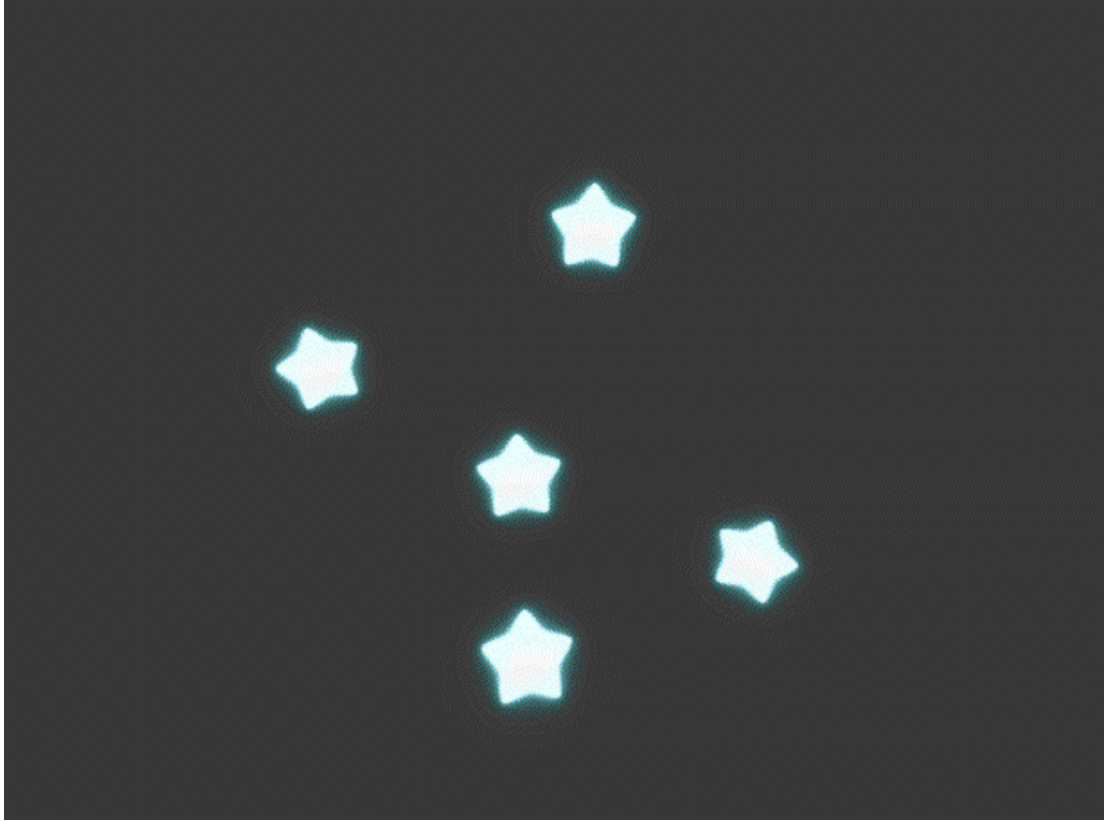


▲ 10.2节：使用GrabPass来实现玻璃效果





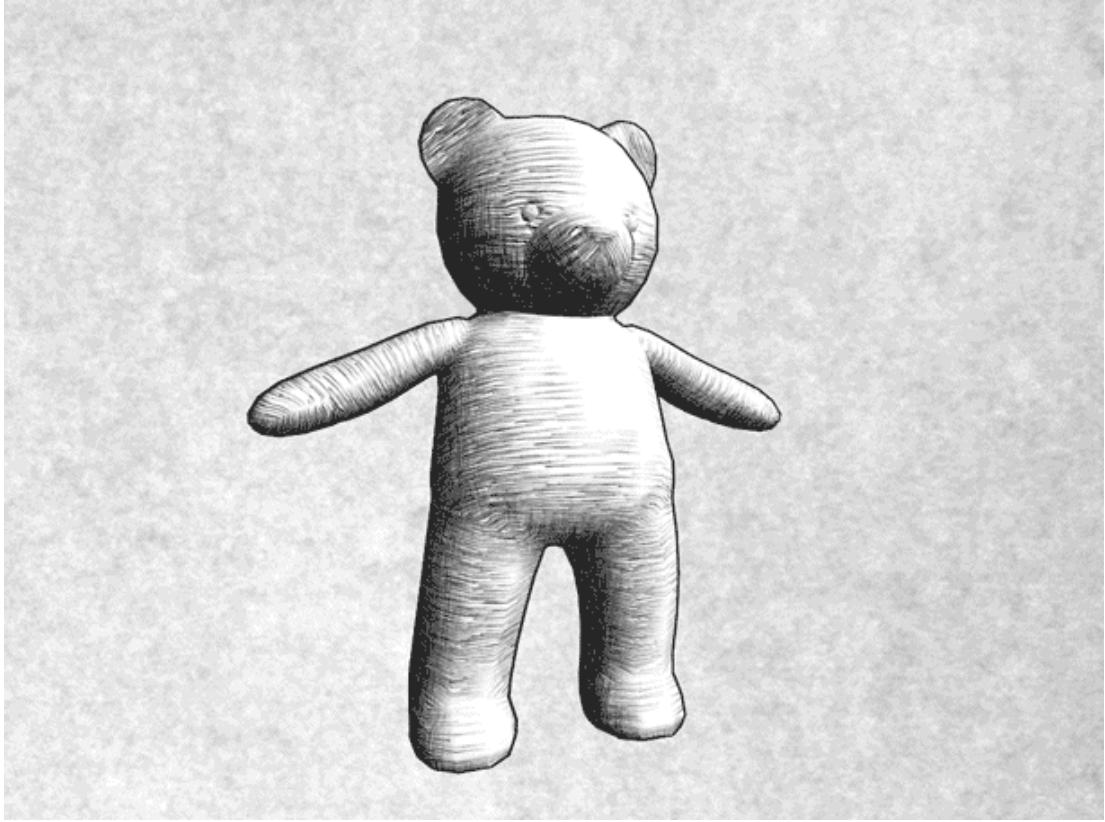
▲ 11.3.1节：使用顶点动画来模拟2D河流



▲ 11.3.2节：广告牌效果

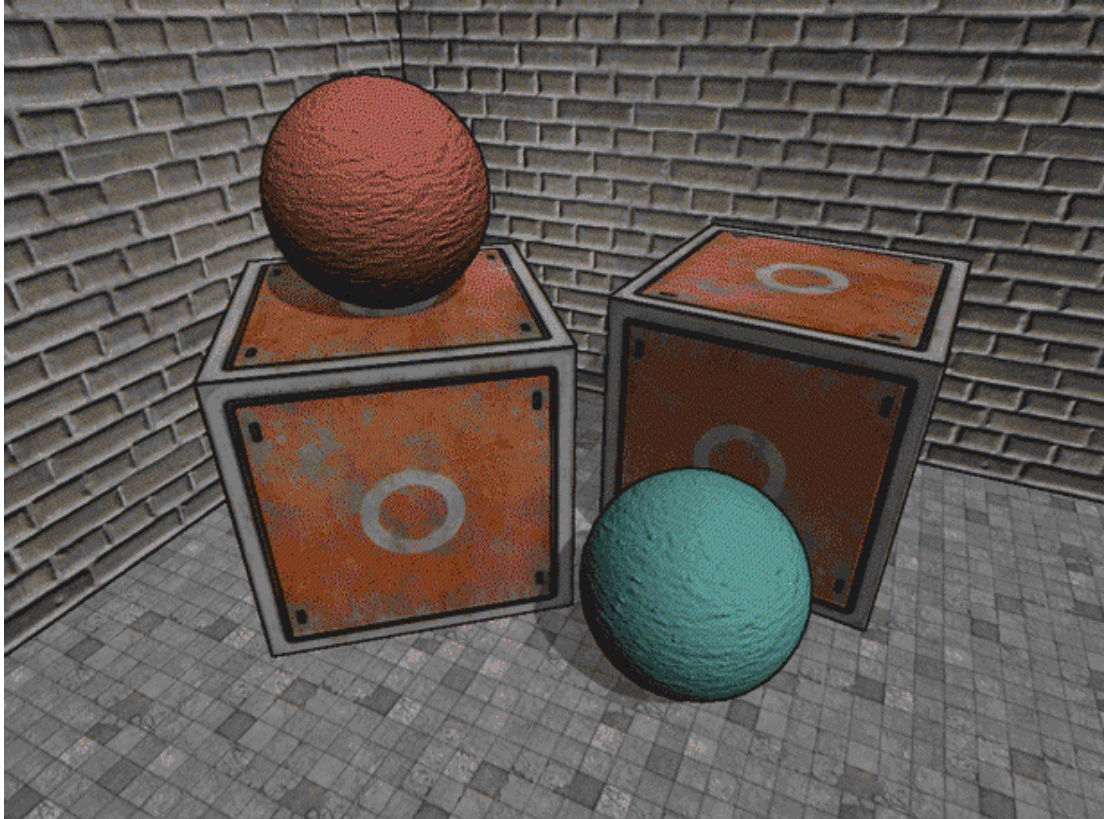


▲ 12.3节：使用边缘检测来实现基本的描边效果

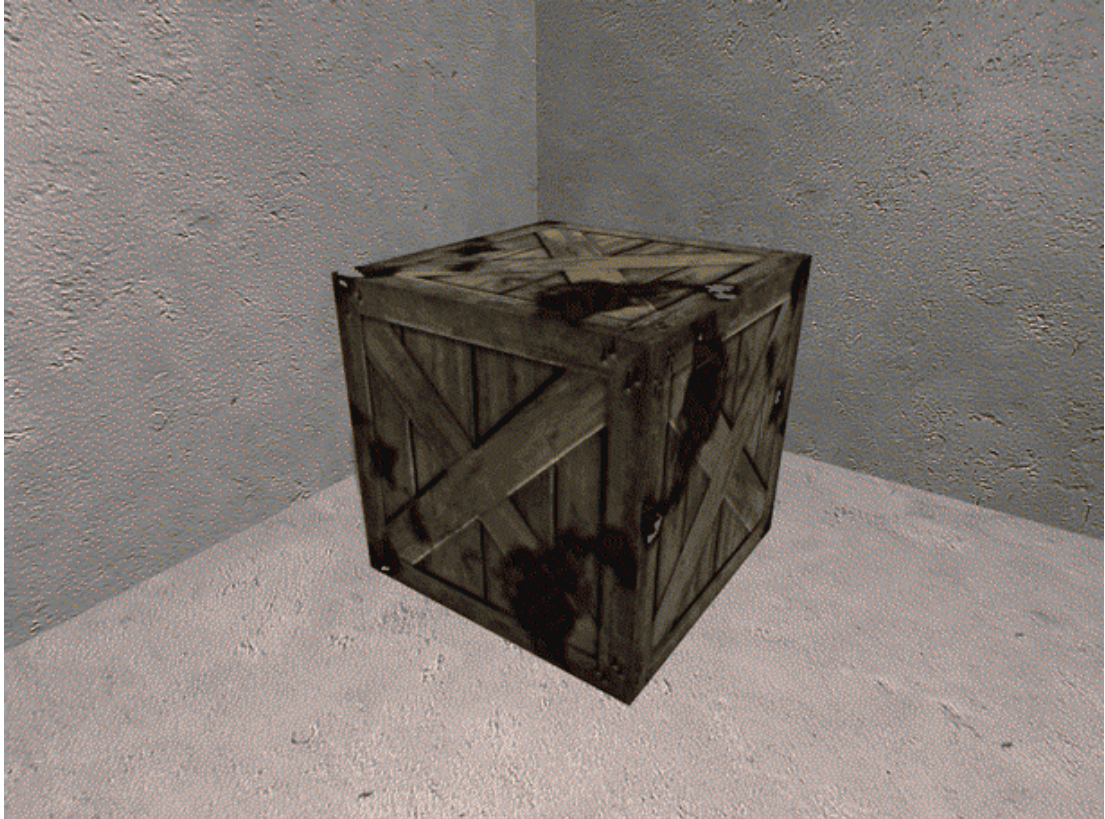


▲ 14.2节：素描风格的渲染



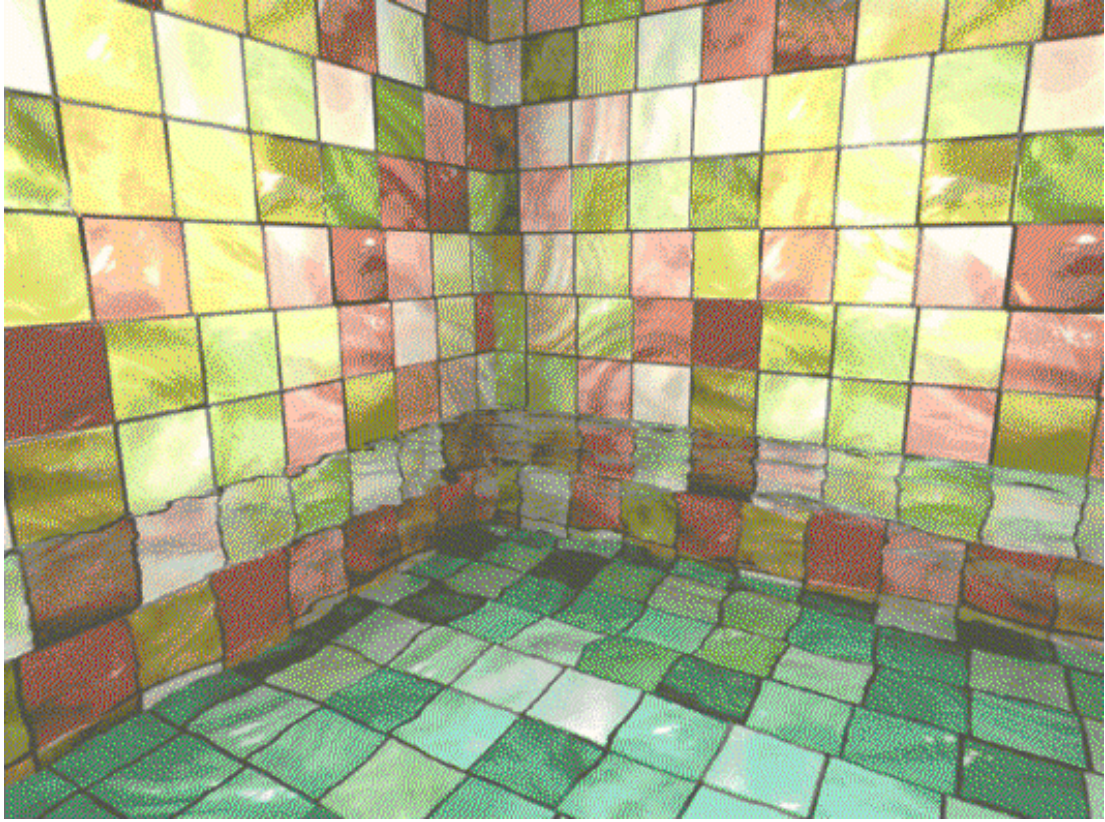


▲ 13.4节：使用深度+法线纹理来实现更加高级的描边效果

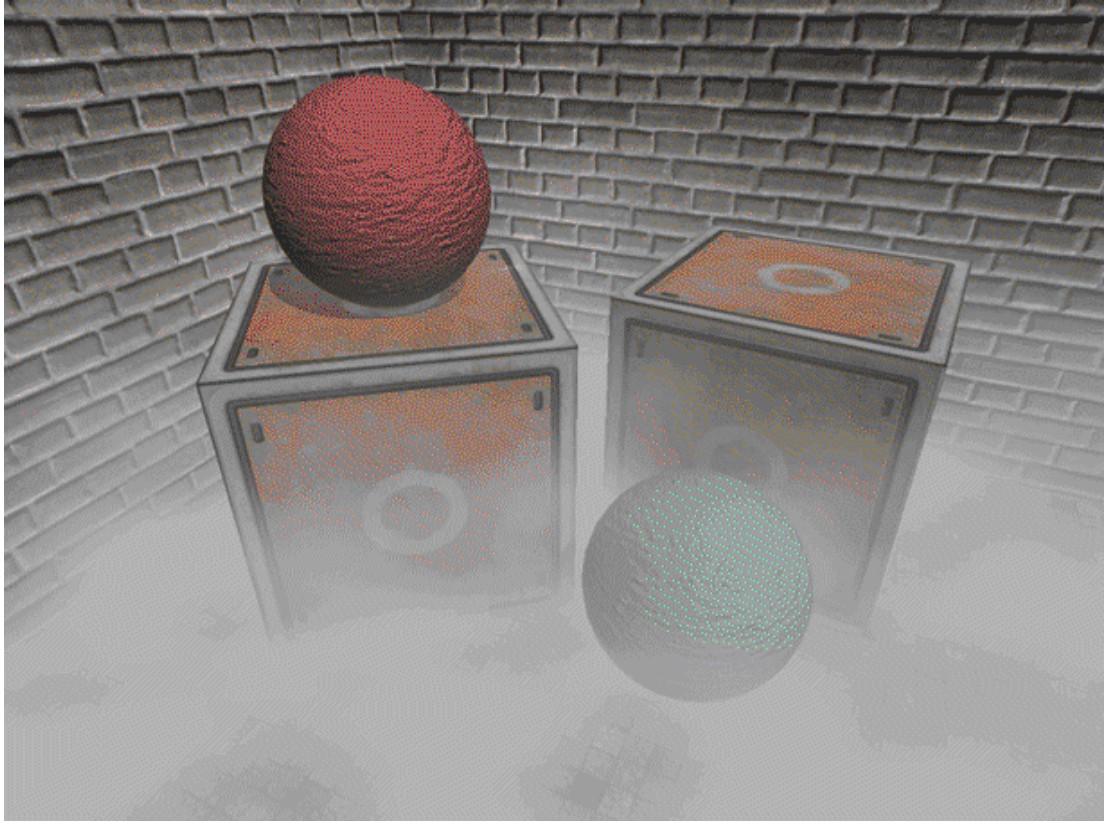


▲ 15.1节：使用噪声纹理来实现消融效果





▲ 15.2节：使用噪声纹理来实现水波效果

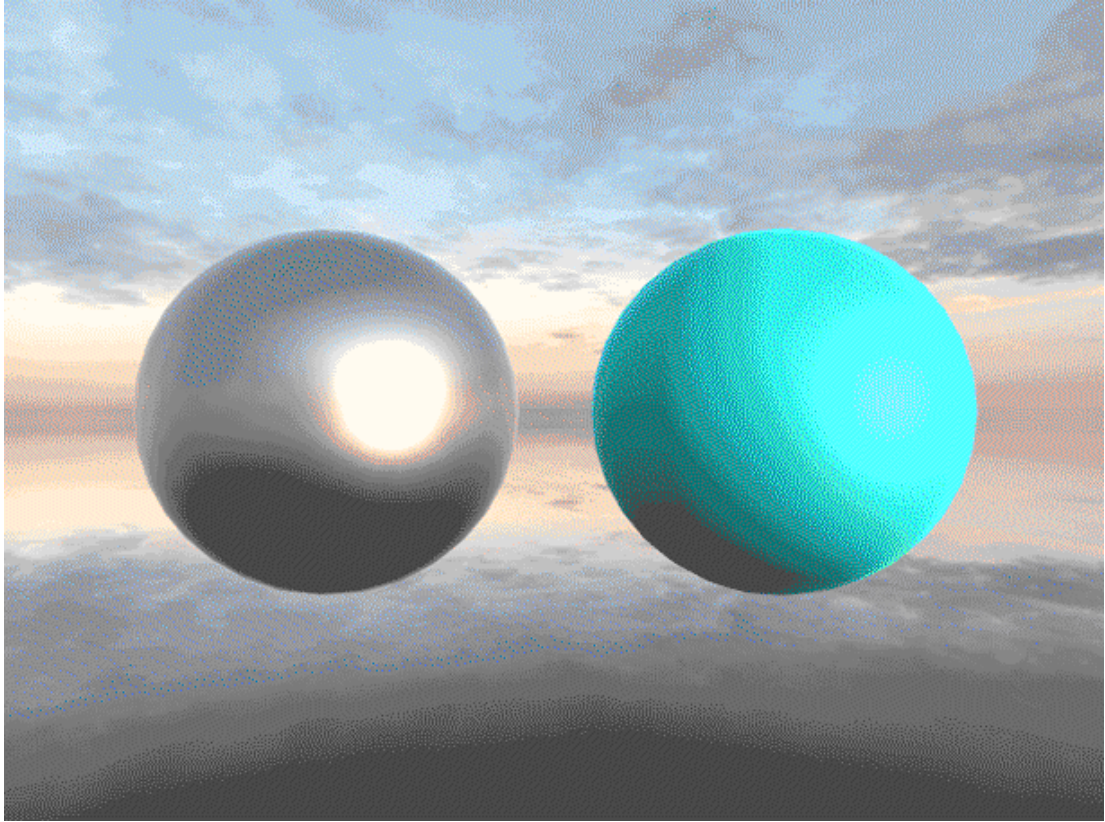


▲ 15.3节：使用噪声纹理来实现非均匀雾效



▲ 17.1节：表面着色器





▲ 18.2节：基于物理的渲染

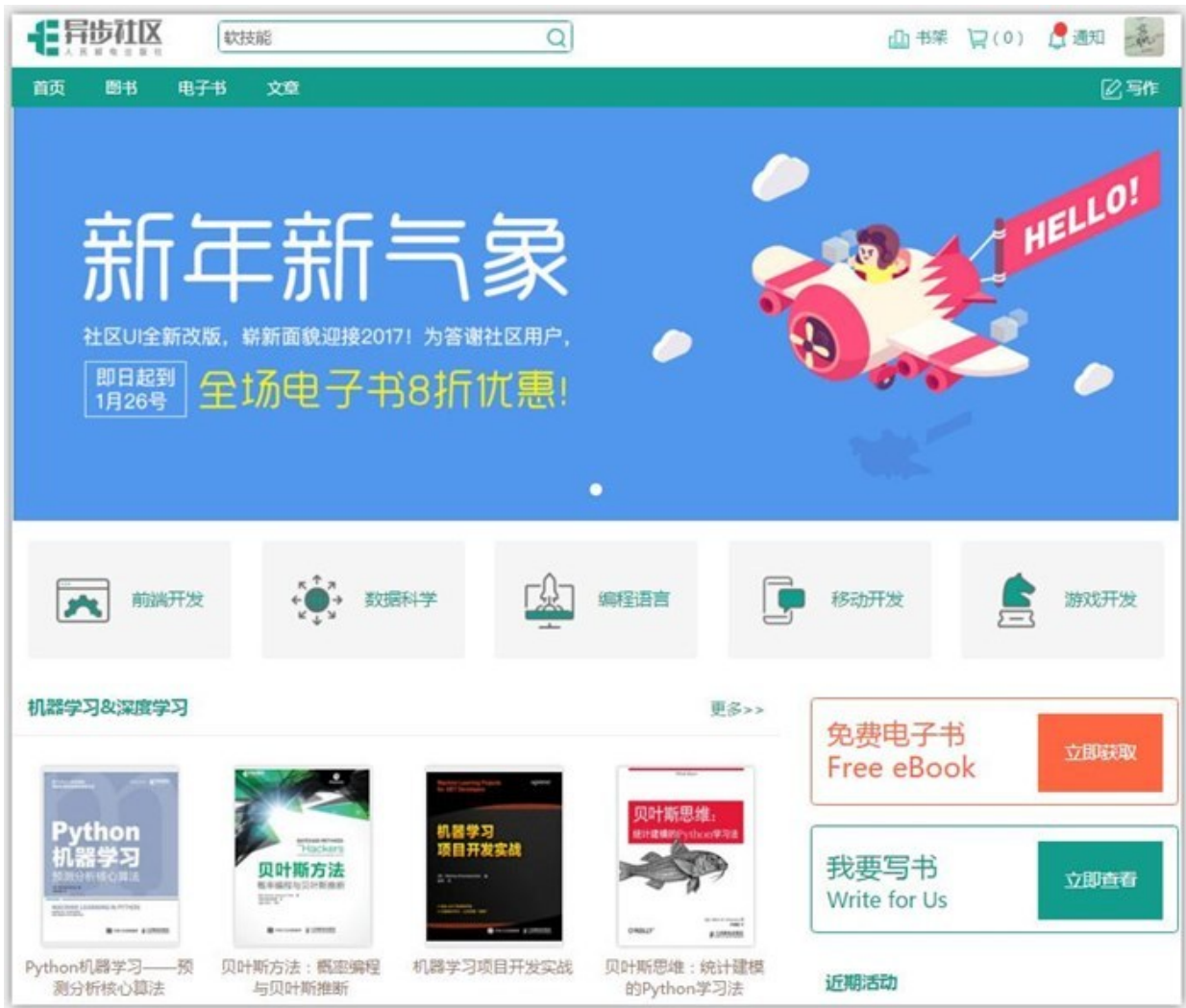
# 欢迎来到异步社区！

## 异步社区的来历

异步社区([www.epubit.com.cn](http://www.epubit.com.cn))是人民邮电出版社旗下IT专业图书旗舰社区，于2015年8月上线运营。

异步社区依托于人民邮电出版社20余年的IT专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与POD按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。





## 社区里都有什么？

### 购买图书

我们出版的图书涵盖主流IT技术，在编程语言、Web技术、数据科学等领域有众多经典畅销图书。社区现已上线图书1000余种，电子书400多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

### 下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

## 与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

## 灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。100积分=1元，购买图书时，在  里填入可使用的积分数值，即可扣减相应金额。

### 特别优惠

购买本电子书的读者专享**异步社区优惠券**。使用方法：注册成为社区用户，在下单购书时输入“**57AWG**”，然后点击“使用优惠码”，即可享受电子书8折优惠（本优惠券只可使用一次）。

## 纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。

The screenshot displays the book page for "Wireshark网络分析的艺术" (The Art of Network Analysis with Wireshark). The page includes the book cover, author information (林沛满), and a detailed description. It features a purchase section with options for paper, electronic, or a combination, along with a "一起购买" (Buy Together) button. The right sidebar shows the author's profile (LinPeiman), a "兑换样书" (Exchange Sample Book) section, and a "精彩推荐" (Recommended) section for "Nmap渗透测试指南".

**Wireshark网络分析的艺术**

作者：林沛满  
责编：傅道坤  
分类：计算机科学 > 安全与加密 > 网络安全

Wireshark是当前最流行的网络包分析工具。它上手简单，无需培训就可入门。很多棘手的网络问题遇到Wireshark都能迎刃而解。本书挑选的网络包来自真实场景，经典且接地气。讲解时采用了生活化的

5.6K 浏览 57 想读 7 推荐

下载PDF样章 配套文件下载

分享：

纸质 ¥45.00-¥31.50 (7折) 电子 ¥25.00 电子+纸质 ¥45.00

购买

纸质 (纸质) + 纸质 (纸质) 总价：75.60 一起购买

目录 评论 9 勘误 1 出版信息

作者简介 专业书评 内容提要

**本书作者**

LinPeiman 上海 1.0K经验值

发私信 送积分 关注

《Wireshark网络分析就这么简单》即《Wireshark网络分析的艺术》作者

**兑换样书**

立即兑换 如何赚取积分

**电子书版本**

PDF Epub Mobi

**精彩推荐**

Nmap渗透测试指南 作者：南广明

## 社区里还可以做什么？

### 提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得100积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

### 写作

社区提供基于Markdown的写作环境，喜欢写作的您可以在这一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

## 会议活动早知道

您可以掌握IT圈的技术会议资讯，更有机会免费获赠大会门票。

## 加入异步

扫描任意二维码都能找到我们：



异步社区



微信订阅号



微信服务号



官方微博



QQ群: 436746675

社区网址: [www.epubit.com.cn](http://www.epubit.com.cn)

官方微信: 异步社区

**官方微博：** @人邮异步社区， @人民邮电出版社-信息技术分社

**投稿&咨询：** [contact@epubit.com.cn](mailto:contact@epubit.com.cn)



# Unity 3D NGUI 实战教程

高雪峰 编著

本书获得以下专业社区一致推荐



**UI中国** UI.cn

中国排名第一的专业界面设计平台



**GAMEUI** Gameui.com

中国专精于游戏视觉相关的设计分享门户网站



**icon** icon.cn

中国专业的游戏 UI 设计咨询机构

 **人民邮电出版社**  
POSTS & TELECOM PRESS

# 目录

[前言](#)

[第1章 初识NGUI](#)

[1.1 游戏UI开发介绍](#)

[1.1.1 什么是游戏UI](#)

[1.1.2 UI为何如此重要](#)

[1.1.3 UI开发的流程](#)

[1.1.4 UI开发的难点](#)

[1.2 什么是NGUI](#)

[1.2.1 NGUI插件介绍](#)

[1.2.2 NGUI的强大优势](#)

[第2章 NGUI基础](#)

[2.1 导入NGUI插件](#)

[2.1.1 NGUI版本介绍](#)

[2.1.2 NGUI的下载和购买](#)

[2.1.3 导入NGUI插件应用](#)

[2.1.4 导入常见问题](#)

[2.2 认识基本的UI资源](#)

[2.2.1 什么是UI精灵 \(Sprite\)](#)

[2.2.2 什么是UI图集 \(Atlas\)](#)

[2.2.3 什么是UI贴图 \(Texture\)](#)

[2.2.4 什么是UI标签 \(Label\)](#)

[2.2.5 什么是UI字体 \(Font\)](#)

[2.3 制作第一个UI图集](#)

[2.3.1 学会解剖UI的资源结构](#)

[2.3.2 如何导入切好的美术资源](#)

[2.3.3 用Atlas Maker制作图集](#)

[2.4 制作第一个UI字体](#)

[2.4.1 为什么要制作UI字体](#)

[2.4.2 静态字体和动态字体](#)

[2.4.3 制作静态字体介绍](#)

[2.4.4 制作动态字体介绍](#)

[2.5 创建第一个UI](#)

[2.5.1 创建一个2D UI](#)

[2.5.2 创建一个3D UI](#)

[2.5.3 了解UIRoot、UIPanel和UICamera组件](#)

[2.6 2DUI和3DUI的工作原理](#)

[2.6.1 2DUI的工作原理](#)

[2.6.2 3DUI的工作原理](#)

[2.6.3 如何判断该选择哪一种UI](#)

[2.7 深度（Depth）概念](#)

[2.7.1 强化对深度的理解](#)

[2.7.2 小心相机的深度](#)

[第3章 核心组件](#)

[3.1 什么是UI控件](#)

[3.2 制作精灵（UISprite）](#)

[3.2.1 怎样判断是否应该使用精灵](#)

[3.2.2 创建精灵](#)

[3.2.3 Sprite组件的设置](#)

[3.3 制作标签（Label）](#)

[3.3.1 怎样判断是否应当使用标签](#)

[3.3.2 创建标签](#)

[3.3.3 Label的文字设置](#)

[3.4 制作UI纹理（UITexture）](#)

[3.4.1 什么情况下使用UITexture](#)

[3.4.2 创建纹理](#)

[3.4.3 纹理的设置](#)

[3.5 制作按钮（Button）](#)

[3.5.1 怎样判断应该使用按钮](#)

[3.5.2 创建按钮](#)

[3.5.3 核心组件BoxCollider](#)

[3.5.4 核心组件UIButton](#)

[3.5.5 制作按钮的放缩动画](#)

[3.5.6 制作按钮的偏移动画](#)

[3.5.7 制作按钮的旋转动画](#)

[3.5.8 添加按钮单击音效](#)

[3.5.9 任何事物都可以变成按钮，不仅仅是UI](#)

[3.6 制作进度条（UISlider）](#)

[3.6.1 怎样判断是否应当使用进度条](#)

[3.6.2 创建进度条](#)

[3.6.3 核心组件UISlider设置](#)

- [3.6.4 进度条的BoxCollider说明](#)
- [3.7 制作输入框（Input）](#)
  - [3.7.1 怎样判断是否应当使用输入框](#)
  - [3.7.2 创建输入框](#)
  - [3.7.3 核心组件Input设置](#)
  - [3.7.4 输入框使用的一些注意事项](#)
- [3.8 制作滚动视图（ScrollView）](#)
  - [3.8.1 怎样判断是否应当使用滚动视图](#)
  - [3.8.2 创建滚动视图](#)
  - [3.8.3 滚动视图核心组件UIPanel](#)
  - [3.8.4 滚动视图核心组件UIScrollView](#)
  - [3.8.5 创建一个拖动条](#)
  - [3.8.6 拖动条说明](#)
  - [3.8.7 让视图内的内容可以被拖动](#)
  - [3.8.8 制作滚动视图时的注意事项](#)
- [3.9 制作复选框（Toggle）](#)
  - [3.9.1 怎样判断是否应当使用复选框](#)
  - [3.9.2 创建复选框](#)
  - [3.9.3 复选框的核心组件UIToggle](#)
- [3.10 制作下拉菜单（PopupList）](#)
  - [3.10.1 怎样判断是否应当使用下拉菜单](#)
  - [3.10.2 创建下拉菜单](#)
  - [3.10.3 显示当前选中的选项](#)
  - [3.10.4 下拉菜单核心组件PopupList](#)
  - [3.10.5 制作下拉菜单的注意事项](#)
- [第4章 UI动画](#)
  - [4.1 常见的两种UI动画介绍](#)
    - [4.1.1 要区分UI动画和UI特效两个概念](#)
    - [4.1.2 关于Tween动画](#)
    - [4.1.3 关于Animation动画](#)
  - [4.2 渐隐渐现动画（透明度动画）](#)
    - [4.2.1 透明度动画的介绍和应用](#)
    - [4.2.2 使用透明度动画TweenAlpha](#)
    - [4.2.3 使用透明度动画的注意点](#)
  - [4.3 颜色变化动画（变色动画）](#)
    - [4.3.1 变色动画的介绍和应用](#)
    - [4.3.2 使用颜色动画TweenColor](#)

- [4.3.3 使用颜色动画的注意点](#)
- [4.4 位置变换动画（位移动画）](#)
  - [4.4.1 位移动画的介绍和应用](#)
  - [4.4.2 使用位移动画TweenPosition](#)
  - [4.4.3 使用位移动画的注意点](#)
- [4.5 旋转变换动画（旋转动画）](#)
  - [4.5.1 旋转动画的介绍和应用](#)
  - [4.5.2 使用旋转动画TweenRotation](#)
  - [4.5.3 使用旋转动画的注意点](#)
- [4.6 大小变化动画（放缩动画）](#)
  - [4.6.1 放缩动画的介绍和应用](#)
  - [4.6.2 使用放缩动画TweenScale](#)
  - [4.6.3 使用放缩动画的注意点](#)
- [4.7 Tween动画总结](#)
- [4.8 动画控制组件UIPlayTween](#)
  - [4.8.1 为什么要用UIPlayTween](#)
  - [4.8.2 动画核心组件UIPlayTween讲解](#)
  - [4.8.3 使用UIPlayTween的注意事项](#)
- [4.9 动画控制组件UIPlayAnimation](#)
  - [4.9.1 为什么要用UIPlayAnimation](#)
  - [4.9.2 为UI添加Animation组件](#)
  - [4.9.3 动画核心组件UIPlayAnimation讲解](#)
  - [4.9.4 使用UIPlayAnimation注意事项](#)
- [第5章 其他组件](#)
  - [5.1 使用Toggle制作页签](#)
    - [5.1.1 页签的工作原理](#)
    - [5.1.2 一个完整的页签界面](#)
    - [5.1.3 制作两个页签按钮](#)
    - [5.1.4 使用ToggleObjects来记录页签内容](#)
    - [5.1.5 制作页签注意事项](#)
  - [5.2 拖动摄像机来浏览超大界面](#)
    - [5.2.1 拖动相机功能的介绍和应用](#)
    - [5.2.2 核心原理和组件介绍](#)
    - [5.2.3 拖动相机浏览超大界面的注意事项](#)
  - [5.3 使用Grid自动排列UI](#)
    - [5.3.1 自动排列UI的应用](#)
    - [5.3.2 自动排列UI核心组件Grid介绍](#)

[5.4 使用DragObject直接拖动物体](#)

[5.5 让玩家通过拖动自由改变控件大小](#)

[5.6 制作序列帧精灵动画（SpriteAnimation）](#)

[5.6.1 什么是序列帧精灵动画](#)

[5.6.2 SpriteAnimation组件](#)

[第6章 NGUI实战进阶](#)

[6.1 UI开发核心问题——UI随屏幕自适应](#)

[6.1.1 屏幕分辨率对UI适配的影响](#)

[6.1.2 主流设备的屏幕分辨率](#)

[6.1.3 自适应核心组件Anchor的使用](#)

[6.1.4 使用Anchor的注意事项](#)

[6.1.5 正式开发UI之前必须明确的几个问题](#)

[6.2 UI元素的相对自适应](#)

[6.2.1 什么是UI元素的相对自适应](#)

[6.2.2 Anchors的介绍及使用](#)

[6.2.3 使用Anchors的范例：背景图的全屏适配](#)

[6.2.4 使用Anchors的注意事项](#)

[6.3 多摄像机同时协作运行](#)

[6.3.1 摄像的渲染层的概念](#)

[6.3.2 多摄像机协作的应用范围](#)

[6.3.3 如何创建多个UI摄像机](#)

[6.3.4 多摄像机协作的注意事项](#)

[6.4 巧用九宫格以减少UI资源量](#)

[6.4.1 项目安装包大小对项目的影](#)

[6.4.2 UI资源量对资源包大小和内存的影响](#)

[6.4.3 什么是九宫格UI](#)

[6.4.4 如何让美术提供合适的九宫格UI资源](#)

[6.4.5 如何在NGUI中划分九宫格](#)

[6.4.6 如何使用九宫格UI](#)

[6.4.7 去掉Mipmap以进一步降低资源包大小和内存占用](#)

[6.5 实战开发中UI资源制作标准](#)

[6.5.1 为什么要设定UI资源制作标准](#)

[6.5.2 资源制作标准设定建议](#)

[6.5.3 程序如何保证UI资源的分辨率不失真](#)

[6.5.4 针对各大平台设置单独的尺寸和格式](#)

[6.6 UI事件监听的击穿](#)

[6.6.1 什么是UI事件监听的击穿](#)

- [6.6.2 如何避免和解决UI事件监听的击穿](#)
- [6.6.3 事件监听遮挡的妙用](#)
- [6.7 开发之前的思考——UI结构设计](#)
- [6.7.1 什么是UI结构设计](#)
- [6.7.2 UI结构设计遵循的一些要点](#)
- [6.7.3 需要的时候，分场景以减轻内存负担](#)
- [第7章 用代码深度控制UI](#)
- [7.1 代码操作NGUI的原理](#)
- [7.1.1 物体与组件的概念](#)
- [7.1.2 怎样用代码操作NGUI](#)
- [7.1.3 获取组件的几种方法](#)
- [7.1.4 迅速判断可以修改的成员](#)
- [7.2 动态加载UI元素](#)
- [7.2.1 为什么游戏中会用到动态加载UI元素](#)
- [7.2.2 擅用UI元素的Prefab](#)
- [7.2.3 将一个物体设置为另一个物体的子物体——  
NGUITools.AddChild\(\)方法](#)
- [7.2.4 NGUITools.AddChild\(\)和Instantiate的区别](#)
- [7.3 擅用EventDelegate事件委托](#)
- [7.3.1 什么是EventDelegate事件委托](#)
- [7.3.2 事件委托的用法](#)
- [7.3.3 哪些地方可以使用事件委托](#)
- [7.4 巧用EventTrigger组件](#)
- [7.4.1 什么是EventTrigger组件](#)
- [7.4.2 EventTrigger用法](#)
- [7.5 常用组件的功能调用](#)
- [7.5.1 UILabel](#)
- [7.5.2 UISprite](#)
- [7.5.3 UITexture](#)
- [7.5.4 UIButton](#)
- [7.5.5 UIGrid](#)
- [7.5.6 UISlider](#)
- [7.5.7 UIToggle](#)
- [7.5.8 UIInput](#)
- [7.5.9 UIPanel](#)
- [7.5.10 UICamera](#)
- [7.6 动画的控制](#)



[7.6.1 为什么要把动画单独提取出来](#)

[7.6.2 控制Tween动画](#)

[7.6.3 关于PlayTween和PlayAnimation](#)

[第8章 实用案例演示](#)

[8.1 角色头像状态栏制作](#)

[8.1.1 示意图和需求分析](#)

[8.1.2 设计并制作UI](#)

[8.1.3 设计并编写代码](#)

[8.2 场景加载的进度条界面制作](#)

[8.2.1 为什么要做这个界面](#)

[8.2.2 异步加载的概念](#)

[8.2.3 制作一个单独的加载界面场景](#)

[8.2.4 设计并编写代码](#)

[8.3 技能快捷栏的制作](#)

[8.3.1 示意图和需求分析](#)

[8.3.2 设计并制作UI](#)

[8.3.3 设计并编写代码](#)

[8.4 角色头顶血条的跟随](#)

[8.4.1 角色头顶血条的跟随分析](#)

[8.4.2 制作血条的UI](#)

[8.4.3 设计并编写代码](#)

[8.5 NGUI多语言切换的实现](#)

[8.5.1 什么是本地化](#)

[8.5.2 NGUI本地化的原理](#)

[8.5.3 本地化案例演示](#)

[第9章 常见疑难问题解答](#)

[9.1 关于NGUI版本问题](#)

[9.2 导入NGUI资源包出错](#)

[9.3 如何创建两个UIRoot](#)

[9.4 如何让粒子在界面上正确显示](#)

[9.5 为什么在父物体上增加透明度动画，子物体没有跟着变化](#)

[9.6 为什么动画播放一遍之后无法再次正常播放](#)

[9.7 为什么3DUI模式下，UI资源的尺寸Snap后和屏幕的大小比例不一致](#)

[9.8 为什么UI不受灯光影响](#)

[9.9 为什么3D模型放到UIRoot下就变得看不见了](#)

[9.10 为什么UI单击后无法播放音效](#)

[9.11 为什么Depth更大的图片反而被Depth小的图片遮住](#)

[9.12 怎样判断点中的东西是UI](#)

[9.13 为什么Label的文字始终不够清晰、明亮](#)

[9.14 为什么创建的物体有BoxCollider却无法接收事件](#)

[9.15 为什么改变了控件的父物体，导致了显示层级错乱](#)

[9.16 关于ScrollView滑动的问题](#)

[笔记栏](#)

[返回总目录](#)

# 版权信息

书名：Unity 3D NGUI实战教程

ISBN：978-7-115-38546-8

本书由人民邮电出版社发行数字版。版权所有，侵权必究。

您购买的人民邮电出版社电子书仅供您个人使用，未经授权，不得以任何方式复制和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

- 编著 高雪峰

责任编辑 张 涛

- 人民邮电出版社出版发行 北京市丰台区成寿寺路11号

邮编 100164 电子邮件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

- 读者服务热线: (010)81055410

反盗版热线: (010)81055315

# 内容提要

NGUI 是专门针对 Unity 引擎、用 C#语言编写的一套插件，它已经成为了目前世界上应用最广、最成熟的Unity制作UI的插件，完美地弥补了Unity引擎原生GUI系统和NewGUI系统的各种不足。程序员可以利用它提供的一整套UI框架和事件通知系统来进行自己项目的UI设计和制作。本书不仅讲解了必知必会的 NGUI 的基础知识，更是以项目实战为目的，涵盖了大量的项目实战中的经验之谈和技巧总结，用以帮助读者达到学以致用目的。

本书的主要内容：初识NGUI、UI开发的流程、NGUI强大优势、制作第一个UI图集、创建一个3D UI、查看和管理UI的深度、制作基础的UI控件、让UI动起来——UI动画、NGUI进阶、使用Panel管理面板、NGUI 实战进阶、UI 开发核心问题——UI随屏幕自适应、实战开发中 UI资源制作标准、跨平台制作UI资源、UI结构设计、UI代码的设计和优化、项目案例实战分析、背包界面的制作等核心技术，最后用一章归纳了NGUI常见疑难问题，以便读者遇到问题时可以随时参考。

本书适合新上手的Unity客户端程序员、需要做UI的Unity程序员、想自学Unity做独立游戏开发的人员，以及大专院校相关专业的师生学习用书和培训学校的教材。

# 前言

在手机游戏开发兴起的当下，Unity 3D 引擎依靠其良好的跨平台特性，一跃成为全球第一大引擎，被广泛地使用。越来越多的游戏开发者开始关注和使用Unity 3D 引擎。Unity 3D引擎最大的短板在于其原生的GUI系统有很大的缺陷，例如，性能和方便程度等都不适合进行商业开发，所以，大部分开发者都开始使用NGUI。NGUI的GUI以良好的性能优化、方便的开发模式、成熟稳定等特点，已经成为全球Unity 游戏开发者的 UI 制作首选插件。因为NGUI是一个插件的缘故，网上很难有成熟的资料，即便要查找一些基本的资料也得连入国外的网站，而且是英文的资料，这给开发者带来了很大的苦恼。由于UI 开发是游戏开发中客户端的重头工作，具有责任大、工作量多、修改频繁等诸多特性，一直是客户端程序员的一个疑难问题。现在市面上还没有NGUI的书，为了让读者能够学会NGUI的用法和技巧，并且能够直接运用于正式的商业项目开发中，作者结合自己的实践经验特意撰写了本书。

## 本书的特点

本书不仅讲解了必知必会的基础知识，更是以项目实战为目的，书中涵盖了大量的项目实战中的经验之谈和技巧总结，这对于一个客户端程序员来说，不管他用什么引擎、用什么 UI工具来开发UI系统，本书都能给他提供帮助，达到学以致用用的目的。

## 本书的主要内容

本书全面讲解了NGUI的实战知识，主要内容为：初识NGUI、UI开发的流程、NGUI强大优势、导入NGUI插件、认识UI的基本资源、制作第一个UI图集、用AtlasMaker制作图集、制作第一个UI字体、创建一个3D UI、3D UI的工作原理、查看和管理UI的深度、制作基础的UI控件、精灵的创建、制作UI纹理、制作按钮、制作进度条、制作滑动条、制作输入框、制作滚动视图、制作复选框、让UI动起来——UI动画、颜色变化动画、位置变化动画、旋转变换动画、大小变化动画、组件整体变换、音量变化动画、在UI中使用Animation动画、动画控制——UIPlayTween组件、NGUI进阶、使用Panel管理面板、使用Grid排列元素、使用Toggle制作页签、使用DragCamera直接拖动摄像机、使用DragObject直接拖动物体、拖动改变UI元素的尺寸、按钮绑定快捷键、制作列表、打字机慢慢出字的效果、NGUI实战进阶、UI开发核心问题——UI随屏幕自适应、背景图的适配、UI元素的相对自适应、多个摄像机同时协作运行、实战开发中UI资源制作标准、跨平台制作UI资源、巧用九宫格减少UI资源量、UI事件监听的遮挡、NGUI和模型、特效在同一层中混用、UI结构设计、用代码操作NGUI的类、获取NGUI的组件、迅速判断类中可读写的成员、动态创建UI元素、Sprite的常用操作、动态对 Button 设置单击事件、网格动态增减成员和刷新排列、手动控制动画随意播放、调用进度条、巧用EventTrigger 监听各种事件、UI 代码的设计和优化、项目案例实战分析、场景加载的进度条界面制作、RPG 游戏中人物头像状态栏的制作、技能快捷栏的制作、RPG角色头顶跟随血条的制作、背包界面的制作等核心技术，最后用一章归纳了NGUI常见疑难问题，以便读者遇到问题时可以随时参考。

## 本书作者

高雪峰，现为游戏制作人，曾经担任游戏主策划、游戏运营、Unity程序员等职位，开发过端游、页游、手游等项目，带团队做过多



个商业项目，对游戏的研发过程具有丰富的经验和实战技能。对NGUI有深入的研究，并且全部应用于项目实战，本书是作者多年实战经验的总结，定会给读者带来很多有益的实战启示。

### **本书读者对象**

本书适合新上手的Unity客户端程序员、需要做UI的Unity程序员、想自学Unity做独立游戏开发的人员阅读，也适合用作大专院校相关专业的师生学习用书和培训学校的教材。

编辑联系邮箱：zhangtao@ptpress.com.cn。

# 第1章 初识NGUI

## 1.1 游戏UI开发介绍

### 1.1.1 什么是游戏UI

UI全称是User Interface，即用户界面。UI的概念运用于各行各业，例如电脑操作系统、手机、网站等，几乎所有需要用户自主操作的地方都会涉及用户界面。UI的职责是负责人机之间的交互，它需要将关键信息、操作逻辑等展示给用户。好的UI设计不但能使操作变得易于理解、简单易用，而且能做到简洁美观，给用户带来舒适的操作体验。在软件设计中，UI的设计是核心设计工作之一。

游戏是一种非常典型的互动娱乐，不论在什么游戏中，玩家都需要进行自己的操作，而且频率非常高。在一些大型游戏中甚至会出现更复杂而频繁的操作，例如竞技游戏和大型网络游戏等。这些在游戏中进行的操作，让玩家体会到了游戏的乐趣，而这一切都依赖于一整套游戏的UI机制。

在游戏中，UI 几乎无处不在，例如进入游戏的菜单、输入账号密码的登录界面、创建和选择角色、角色血条状态栏和技能栏、场景雷达图、各种各样的提示框等。

### 1.1.2 UI为何如此重要

在游戏中UI主要承担了以下职责：

### 1. 游戏美术风格的重要组成部分

一个游戏由不同种类的美术资源组成，例如角色、场景、特效、UI 等。每一个游戏都必须有一个统一的美术风格，否则游戏将变成不伦不类的四不像。例如中国仙侠风格的游戏，不应该出现欧美魔幻的元素；同理，欧美魔幻风的游戏里，不应该出现中国仙侠的元素。UI 因其无处不在的特性，成为游戏设定美术风格时一个非常重要的考量指标。

### 2. 承担着重要的美观职责

当玩家打开游戏看到第一个游戏画面时，UI 就将一直陪伴着玩家直到玩家退出游戏。甚至大部分UI都是长久性地出现在玩家的游戏窗口中，例如玩家角色的状态栏、技能栏等。所以，UI的美观和耐看，将会直接影响着玩家对这款游戏的美观程度的评价。

### 3. 负责清晰明了地展现游戏的操作方式

每一个游戏都有着自己的操作方式，越是大型游戏，操作逻辑往往会越复杂和庞大。好的UI设计，能让用户不依赖指引就能够迅速地知道自己应该怎样操作来进行游戏。

### 4. 能让玩家舒服、方便地进行人机交互

游戏UI的核心目的就是让玩家能进行自主操作，而游戏的机制往往很复杂，这个时候“人性化”就变得很重要。例如技能冷却完成之后，技能栏闪烁一下，玩家在游戏中不自觉地就能知道技能已经冷却完成。

## **1.1.3 UI开发的流程**

在项目开发中，一般都是由策划人员、程序人员、美术人员、测试人员组成多个组协同进行开发。在开发 UI时，首先应当由美术核心人员确立 UI的整体美术风格标准，然后再进行UI的各个模块的开发。

在开发 UI 的模块时，首先应当由相关策划人员提出功能的需求，指出该模块的UI应该具备什么功能、能让玩家进行哪些操作逻辑。然后，由策划人员给出UI的控件布局图或者由美术人员自行设计布局图。再由UI美术人员进行UI的资源制作，制作完成后，将UI资源提交给客户端程序员。客户端程序员拿着美术人员提供的UI资源，按照 UI 的布局图和功能文档，最终完成该模块的功能开发。

### 1.1.4 UI开发的难点

UI在开发中往往具备以下几个问题。

- (1) 需求不清晰。
- (2) 功能难实现。
- (3) 工作量很大。
- (4) 优化不够好。
- (5) 改动太频繁。

这些问题都是项目实际开发中非常常见的问题，特别是客户端程序员经常为以上五个问题伤脑筋。要规避它们，除了良好的沟通以外，还需要足够的对UI的思考，以完成一个良好的UI架构来提高抗风险指数，并扩充自身的知识面，以做到提早考虑到更多的特殊情况。

看完本书后，相信你一定能找到应对以上项目开发实际问题的最优方案！

## 1.2 什么是NGUI

### 1.2.1 NGUI插件介绍

NGUI是专门针对Unity引擎、用C#语言编写的一套插件，经历了数十个版本的更迭之后，它已经成为了目前世界上应用最广、最成熟的Unity制作UI的插件，完美地弥补了Unity引擎原生GUI系统和NewGUI系统的各种不足之处。程序员可以利用它提供的一整套UI框架和事件通知系统来进行项目的UI设计和制作，NGUI凭借其强大的功能、良好的优化和易用易学性，让大多数程序员都赞赏有加！

## 1.2.2 NGUI的强大优势

### 1. 成熟稳定

NGUI 经历了数十个版本的更迭，发展到现在几乎没有 BUG，并且完美支持跨平台和自适应。在这一点上，它远远超越了其他的UI插件。论成熟稳定，它比Unity的NewGUI还要更胜一筹。

### 2. 功能丰富

NGUI除了满足普遍的UI制作功能以外，还集成了大量的封装好的实用功能，比如拖曳、更多的事件监听、各种Tween动画、本地化等，甚至支持光照、法线、折射等特性。这是NGUI远远超过其他UI制作方式的地方，也是NGUI经历数十个版本更迭的积累。

### 3. 操作方便

NGUI的操作几乎都集中于Inspector面板中，并且不需要Play运行就能看到UI的结果。各个模块和组件封装非常好，需要功能时只要为其附上相应的NGUI组件就可以完成，大部分功能都不需要自己写代码。

### 4. 极致优化

目前最新的 NGUI 版本中，对于 UI 的渲染性能已经优化到了极致，使用 1 个 DrawCall就能完成绝大部分UI的渲染。

### 5. 高灵活性

NGUI都是以组件形式使用，程序员可以不借助任何外部的资源进行UI的制作，可以让任何一个控件通过改变其组件的方式，自由地变换为按钮、精灵、进度条、输入框等。

## 第2章 NGUI基础

### 2.1 导入NGUI插件

#### 2.1.1 NGUI版本介绍

NGUI插件目前较新的版本是3.6以后的版本。

在NGUI 3.0 以前的时期，底层的事件通信体系完全依赖于SendMessage，这是一个效率比较低下的发消息方式，那个时期大多数Unity的开发者都在使用当时很流行的NGUI 2.6 版本，甚至目前还有少数开发者在使用。

在NGUI 3.0 及以后的版本中，NGUI 进行了大革新，其中革命性的就是将整个的底层消息机制全部换为效率高的EventDelegate。并且开始重视NGUI的性能优化，一直到3.6时期， NGUI相比以前，提高了事件分发的效率，将性能优化到了极致，整合了大量的相近功能，并大大丰富了NGUI的功能类型。

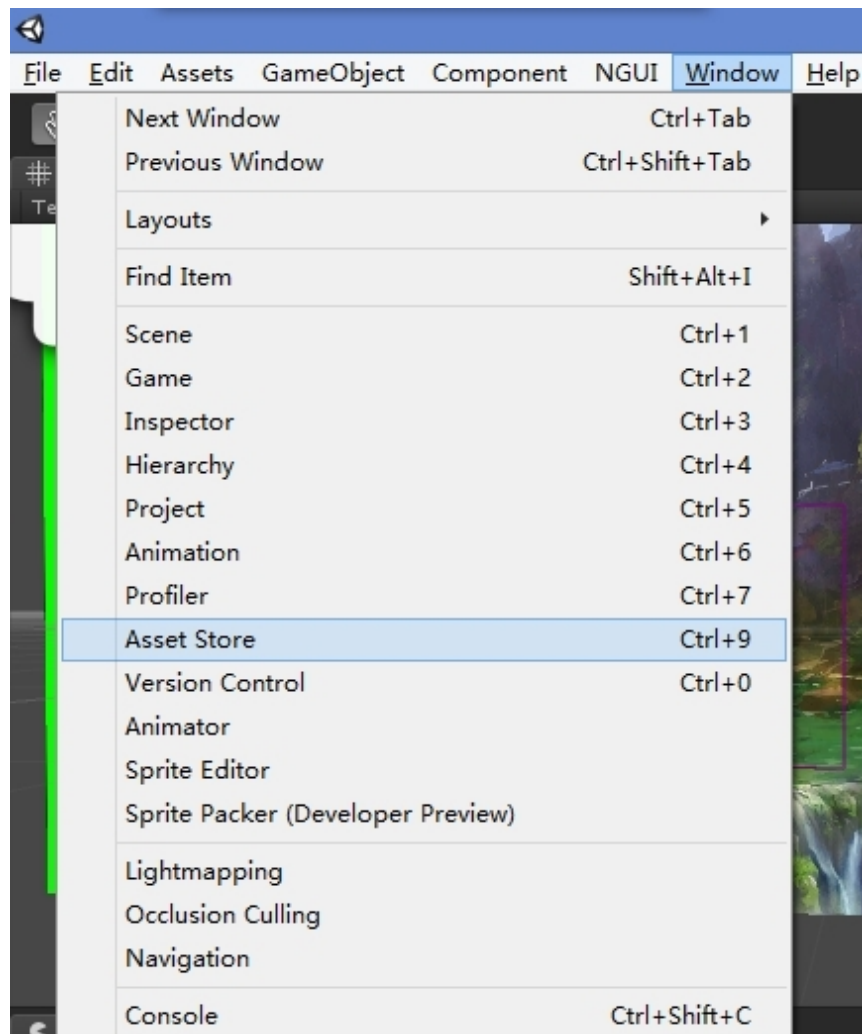
在本书中，将使用比较新的NGUI 3.6.8版本进行讲解，书中可能部分截图并不是3.6.8的版本，但是NGUI 3.6.0 及以后的版本几乎都一样，本书的知识点通用于NGUI 3.6.0 及以后的所有版本，读者大可放心。

#### 2.1.2 NGUI的下载和购买



NGUI本身是一个付费插件，开发者可以从Unity官方的AssetStore（官方提供的Unity资源买卖平台，里面有很多第三方的资源和插件出售，有的收费有的免费）中去购买，售价大约95美金（折合人民币591元）。用户可以从Unity引擎的编辑器界面的顶部Window菜单中选择Asset Store进入，如图2.1所示。也可以在浏览器中输入网址<https://www.assetstore.unity3d.com/>进入。

因为NGUI正版插件价格不菲，如果仅仅是为了学习和练习，开发者可以从网上去下载他人购买的NGUI插件包来使用，效果是一样的。不管是从官方购买的还是自己从第三方网站上下载的NGUI插件，它都应该是一个格式为“.unitypackage”的Unity资源包文件，如图2.2所示。



▲ 图2.1



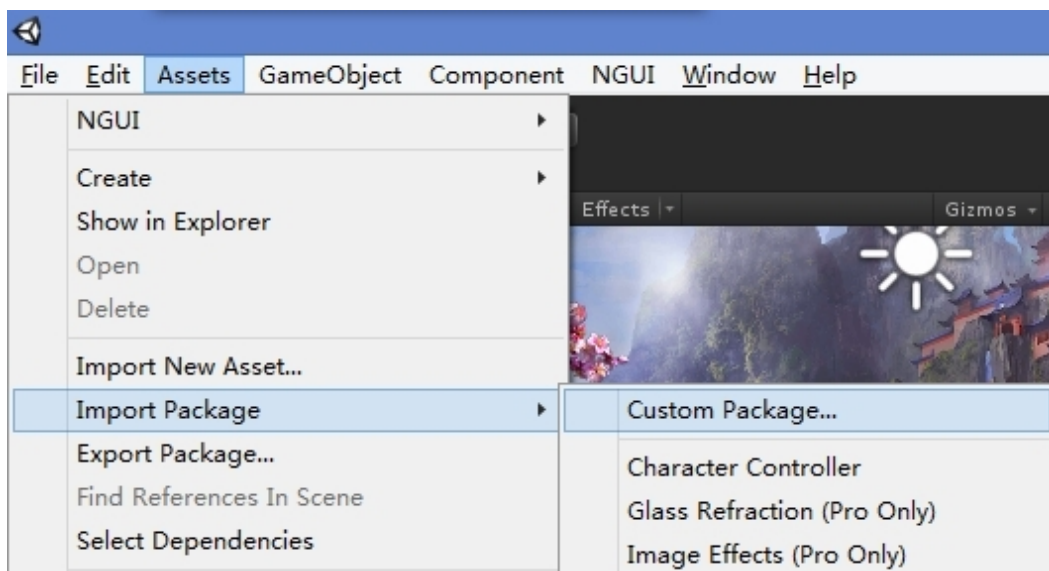
NGUI Next-Gen UI v3.6.8.unitypackage

▲ 图2.2

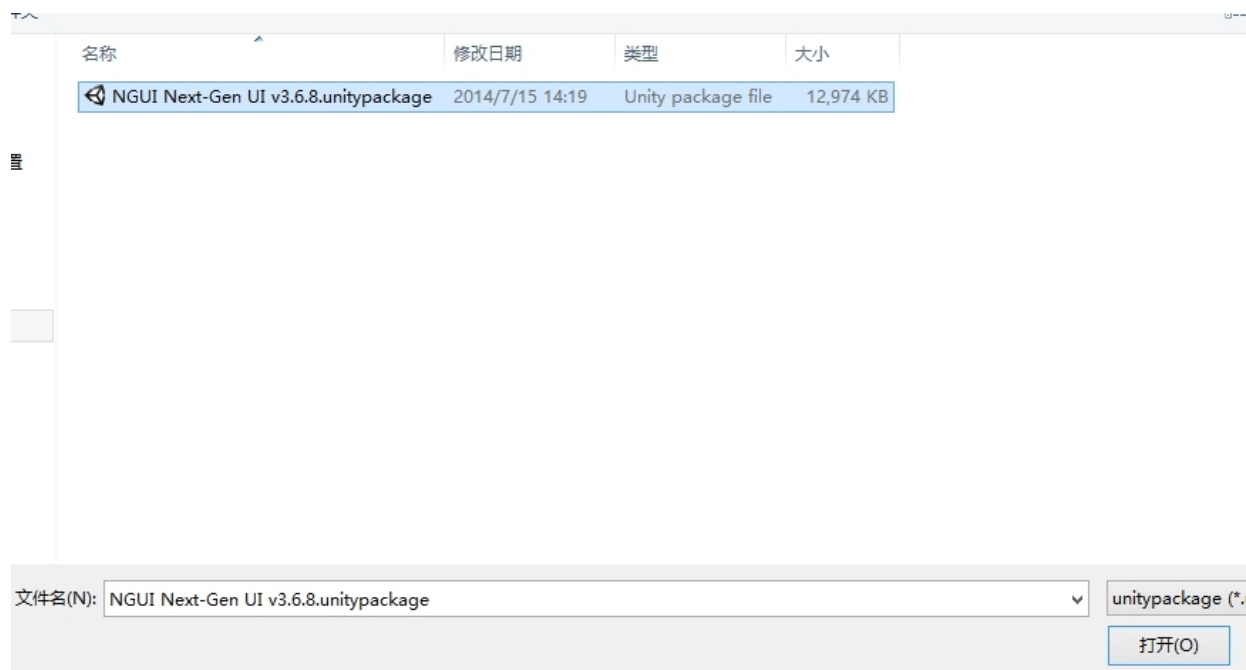
### 2.1.3 导入NGUI插件应用

当下载好NGUI的插件资源包之后，下面要做的就是将NGUI资源包导入到引擎中进行使用。

如图2.3 所示，在Unity编辑器顶部菜单栏中的Assets 菜单中选中 Import Package，然后选择 Custom Package（自定义资源包），弹出图 2.4 所示的资源路径窗口，在其中找到 NGUI资源包所在的位置，单击“打开”按钮即可。



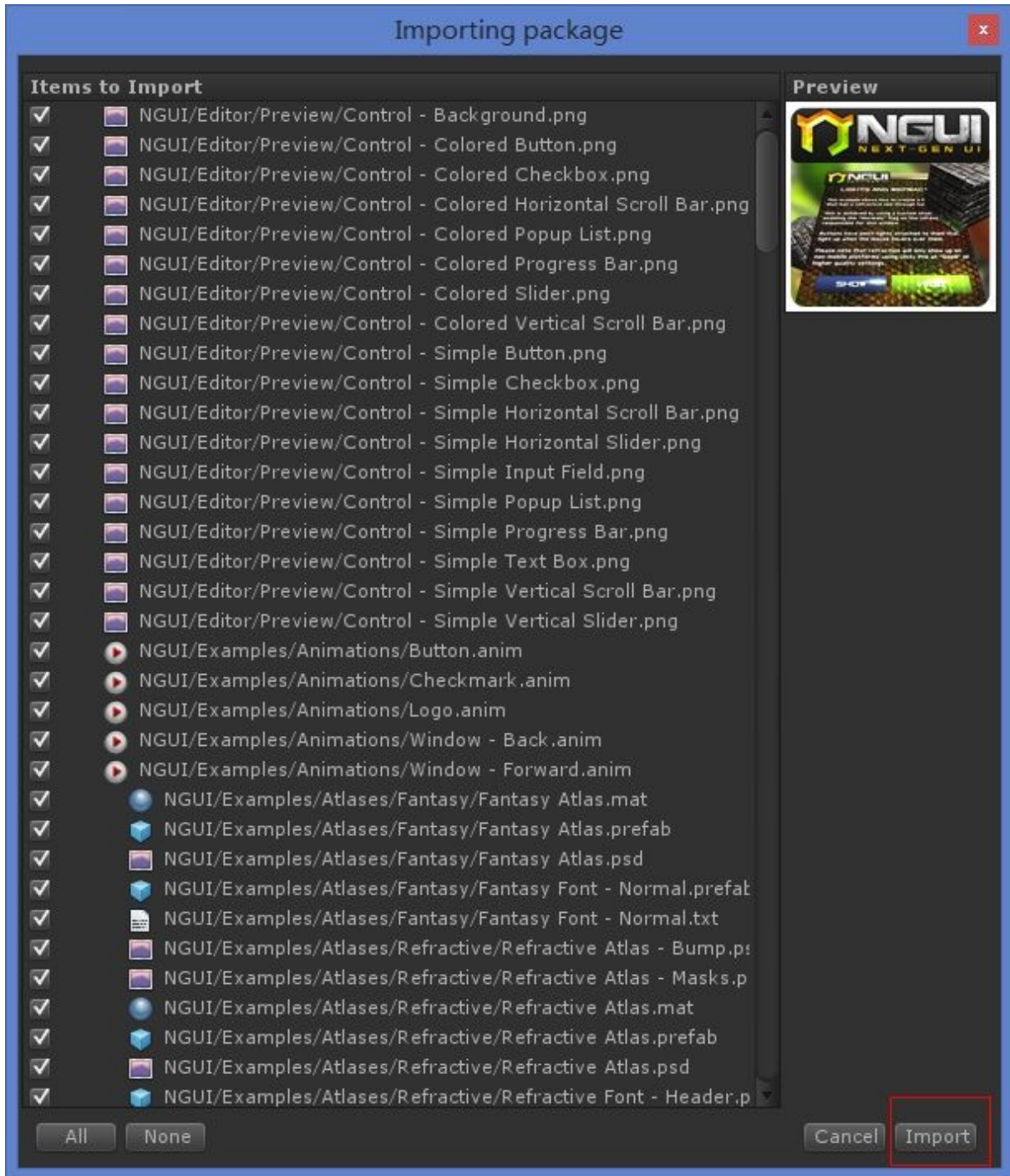
▲ 图2.3



▲ 图2.4

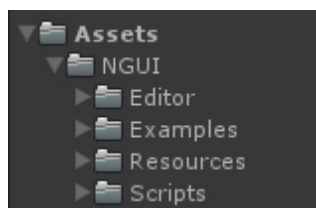
特别注意：请将NGUI资源包放在一个没有中文的路径下再进行导入（例如F:\DownLoad\NGUI-Next-Gen-UI-v3.6.8）。因为 Unity 导入外部资源时，将无法导入带有中文路径的资源，例如F:\下载\NGUI-Next-Gen-UI-v3.6.8因为将NGUI放在了中文名文件夹“下载”之下，将导致NGUI资源包无法成功导入。

单击“打开”按钮后，等待Unity引擎解压资源包，然后将会在Unity引擎界面中弹出图2.5所示的窗口，展示该资源包的内容列表，让用户选择导入哪些资源（默认情况下是全部选择），此时因为其已经默认全部选择，可以直接单击Import按钮，将其全部导入。



▲图2.5

导入成功后，可以看到NGUI文件在Project视图中的结构图（如图2.6所示），其中Editor文件夹是编辑器所用的，不用管它；Examples文件夹是Unity制作的一些基本案例，读者可以从Examples下面的Scenes文件夹中选择它制作的范例场景来参看一些基础功能的制作，如图2.7所示。Resources文件夹存储着NGUI自带范例所用到的资源。最后就是整个NGUI对于开发者来说最核心的文件夹：Scripts，这里面是NGUI已经封装好的各个功能模块的脚本，当需要使用时只要把相应的脚本变成UI物体的组件就可以进行相关的操作了，本书在后文将会介绍更简单的使用方式。

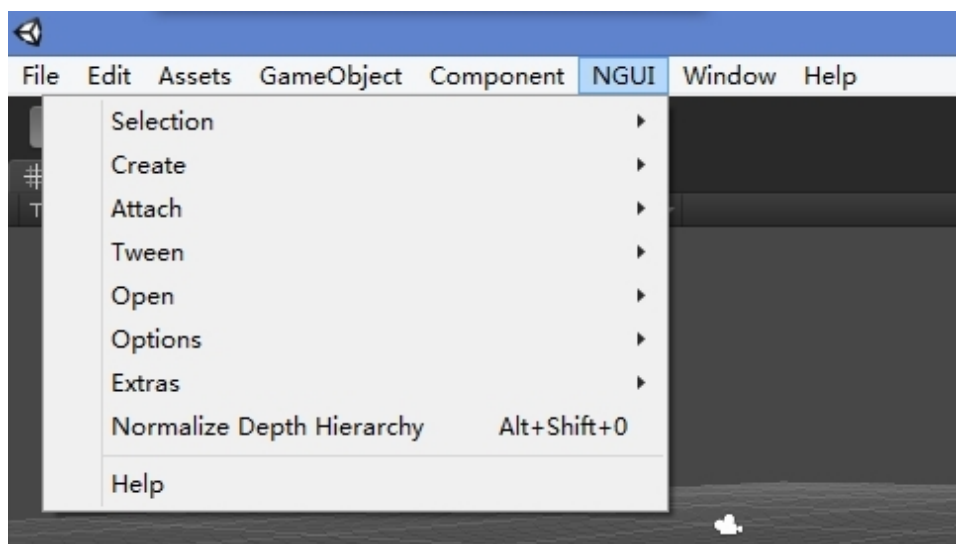


▲图2.6



▲ 图2.7

另外，导入之后还会注意到Unity编辑器顶部菜单栏中多了一项NGUI菜单，如图2.8所示，这个菜单将会是以后使用NGUI制作UI系统最常用的一个菜单。





▲图2.8

好了，到这里就已经说明你已经成功地导入了NGUI插件，这个插件包大约12MB，不过完全不用担心它会让你的项目安装包增大很多，因为NGUI插件包导入后并不存在于Resources文件夹下面，所以在项目工程最后发布时，它只会将NGUI资源包中你所用到的部分纳入打包资源，对项目发布的游戏安装包体积的影响几乎可以忽略不计。

### 2.1.4 导入常见问题

(1) 如果我的工程文件中已经导入过一次NGUI的资源包了，此时我再导入一个新的NGUI资源包会有什么结果？

答：会根据路径替换掉同名文件，并导入额外的新文件。

(2) 导入解压后，并没有弹出图2.5所示的资源包内容预览窗口，而是在Unity编辑器窗口底部报出了一行如图2.9所示的错误，怎么办？



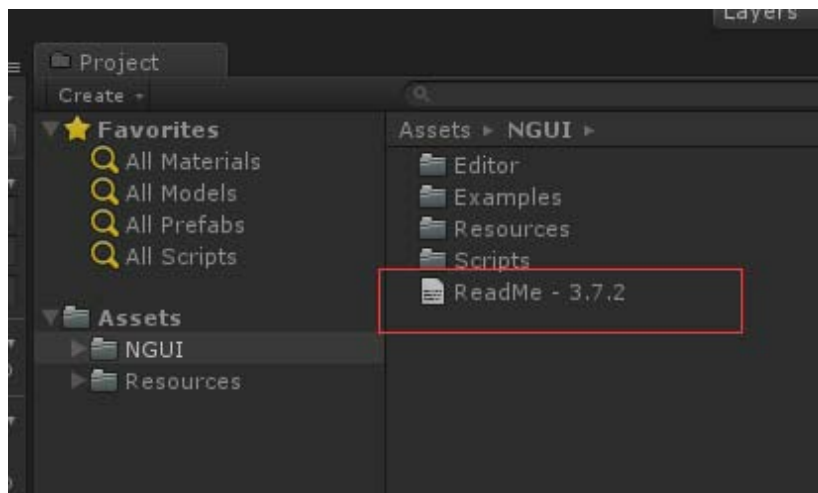
▲图2.9

答：这是因为你将NGUI资源包放在了一个带有中文名称的文件夹路径下，Unity导入任何带有中文路径的资源包时，都会弹出这个错误导致无法导入。请将你要导入的资源包放在一个没有任何中文的路径下再进行导入。

(3) 我怎样查看我的NGUI版本是多少？

答：在Project窗口中，选择Assets文件夹下面的NGUI文件夹，然后会看到一个ReadMe的版本说明文件，这个文件名会带有版本号，如图2.10所示。





▲ 图2.10

## 2.2 认识基本的UI资源

### 2.2.1 什么是UI精灵 (Sprite)

我们在制作 UI时，经常将一些零碎的小的 UI资源（比如，一个小箭头、一个按钮等）打包成一张大图，然后在使用时，只使用这个大图的一部分（例如，只使用其中小箭头的那一小块），那么这一块“被切出来”的图片，就可以称之为精灵。

如图2.11所示的就是一个又一个的UISprite。



▲图2.11

### 2.2.2 什么是UI图集 (Atlas)

我们在制作 UI 时，会将一些零碎的小的 UI 资源打包到一张大图  
中，然后再通过精灵的方式对这张大图进行使用，这张大图就是一个  
图集。这样不但可以减小美术资源的总体积，还可以减少载入内存的  
操作（图集作为一张整图会被一次性载入到内存中）并提高渲染性  
能，而且还可以减少维护大量零碎小资源的麻烦。

如图2.12所示的则是一个由Sprite组成的图集。

### 2.2.3 什么是UI贴图 (Texture)

在NGUI中也有UITexture的概念，这个UITexture从功能用途上和  
Sprite精灵有很大的相似之处，都是为了显示一些图片资源。它和Sprite  
最大的区别在于，UITexture 是一张独立的图，不依托于任何的图集，  
这张Texture有自己的材质球和Shader，每一个UITexture都将消耗一个  
DrawCall（不了解的读者可以理解为一个性能消耗单位）去渲染，每  
一个UITexture都将独立进行加载。

如图2.13中的大背景图就是UITexture。



▲ 图2.12

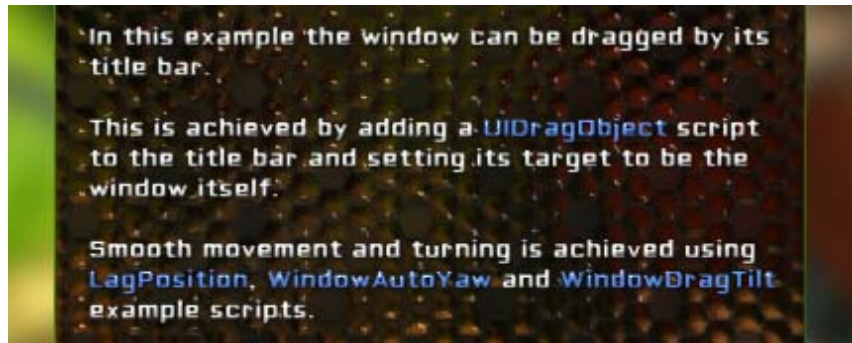


▲图2.13

## 2.2.4 什么是UI标签 (Label)

标签 (Label) 在NGUI中并不是指一种标记物，而是指一种纯文本的UI元素。凡是由程序在UI上打出来的字，都属于标签的内容。例如，如果你需要在界面上长期地显示一行字：请打开背包进行整理，那么这行字属于一个Label。再比如，如果你需要显示角色的生命值为100/200，这个数字会随着角色的生命值而变化，这个生命值的数字也属于一个标签，然后代码会根据角色的血量去读取并改变这个标签的内容。

如图2.14所示框中所有的文字信息都属于Label。



▲图2.14

## 2.2.5 什么是UI字体 (Font)

在制作UI的过程中，不可能所有地方都由美术完成，最典型的例子就是UI上面的文字。很多时候UI上面的文字都是不停地在进行变化，并且没有什么复杂的艺术字效果，不可能全部由美术制作成图片提供给程序，这个时候就需要程序在UI上进行写字。程序在UI上写字时，就将用到UI字体。

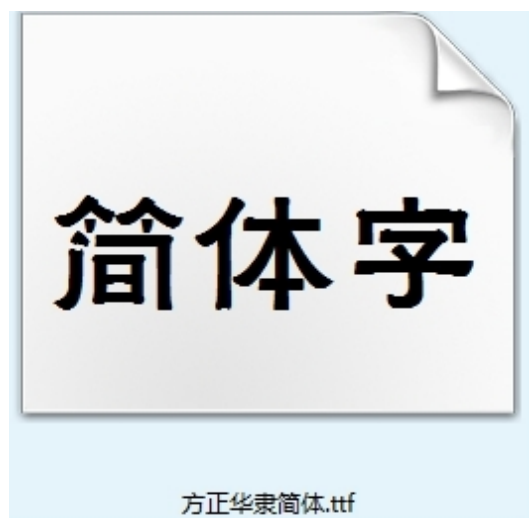


NGUI 的字体分为动态字体和静态字体。程序人员可以选择把某种特殊字体文件中的一些所需的字拿出来形成一张图，然后打字时会从这张图里去调用文字（类似于调用Sprite），这就是静态字体。也可以直接导入字体文件（例如，宋体、楷体等字体文件），打字时只要字体文件里拥有的字都能正常使用，这就是动态字体。当然，NGUI有系统自带的默认动态字体。

如图2.15所示则为静态字体图集，图2.16所示则为动态字体文件（.ttf格式）。



▲ 图2.15



▲ 图2.16

## 2.3 制作第一个UI图集

### 2.3.1 学会解剖UI的资源结构

为什么要剖析UI的资源结构？因为通常情况下，策划设计好UI的功能和大概布局之后，美术人员会做出一张UI的成品效果图，我们称之为UI设计图。然后，客户端程序需要根据自己的制作方式，告诉美术人员如何分割出相应的UI元素提交给程序，以完成制作。



▲ 图2.17

下面以图2.17所示的UI设计图作为例子来讲解分析。首先说明一点，客户端程序一定要同时拿到UI设计图和UI功能描述文档才能彻底知道这个UI会进行什么样的操作逻辑，此时才能准确地对UI资源结构进行剖析。主要有以下一些方法。

- 看设计图，从相关设计文档了解该UI的功能。

在不了解UI功能的情况下，程序只看设计图不一定能准确地明白这个UI的全部用途，为了减少返工，请务必先了解该UI模块的全部功能。如图2.17所示，则是一个典型的登录UI。

- 观察UI的设计图，判断哪些字是程序可以写的，哪些字是程序写不了的。

在图中我们发现“登录”“请输入账号”“请输入密码”等Label。其中，“请输入账号”和“请输入密码”两个标签的文字并没有什么特殊的美术效果，完全可以由程序来完成，就不需要美术提供图片资源了。

而“账号”和“密码”两个标签，虽然本图中可以由程序完成，但是如果碰到那种有特殊美术效果（比如，文字上有华丽的渐变和高光等效果，程序是完成不了的）的，则需要让美术提供一张写有相应文字的图片，用Sprite去代替这个Label。

后文我们会详细讲解什么情况下使用Label。

- 通过设计图判断哪些是Sprite，充分考虑复用性。

凡是零碎的、小的图片资源，都可以是Sprite。如图2.17中，输入账号密码有一个底框，整个界面又有一个底框，这些都可以让美术人员切成单独的Sprite，然后由程序来进行拼装。

在分割Sprite时，尽量分割得细一点，如图2.17中所示，我们可以将整个UI的大底框和输入账号密码的两个底框合并在一起切给程序，但是这样将导致这个UI图片无法用于其他地方。如果整个UI的大底框和输入账号密码的两个小底框分开来切，那么以后其他需要用底框的地方，就可以复用图中切出的UI资源。

相同的UI元素只需要一份就够了，例如图中输入账号的底框和输入密码的底框是一样的，于是只需要切出一份就可以了。

后文我们会详细讲解什么情况下使用Sprite和Texture。

- 通过设计图判断出按钮资源的制作方式。

按钮笼统地分有两种形式，一种是普通按钮，也就是一张没有文字的按钮图片，程序人员在需要用时，就在上面写上“确定”等不同的、当前所需要的文字。另一种按钮则是图片按钮（以前旧版本NGUI的ImageButton），这种按钮的特点是整个按钮就是一张图片，它既是按钮也是图片。



如图2.17中所示，登录和注册两个按钮几乎一模一样，只是上面的文字有区别，而“登录”和“注册”这几个文字明显是程序可以写出的文字效果，于是可以让美术人员切出一张普通按钮资源的底图，然后程序再在上面写字。

后文我们会详细讲解什么情况下使用按钮。

- 通过设计图判断其他控件。

要比较迅速地结合 UI的功能判断出 UI中的一些其他控件，例如假设这是一个人物生命法力的状态栏，那么一定要区分出UI中是否用来显示人物生命法力值的进度条，如果有进度条，则进度条需要上面切一条表层条，下面切一个空槽底板条。

后文我们会对相关知识进行详细说明。

在剖析UI资源结构的时候，有无数种方案，每一种方案UI美术人员基本都能完成，程序人员也能完成相应的功能。但是，我们一定要通过充分的分析去使用一种相对更好的方案，这样对项目以后的修改和拓展都有好处。在剖析UI资源结构时一定要秉承以下几个原则：

- 尽量保证还原设计图的效果，不损失质量，这是前提。
- 尽量发现重复的元件，而且重复的元件只需要一份就足够。
- 尽量分割得零碎一点，避免多个元件合并在一起出图，这样对项目不利（后文会详细讲解）。
- 尽量使用九宫格来制作比较大的底板、底框等（后文会详细讲解）。
- UI切图全部让美术人员以PNG格式导出。

### 2.3.2 如何导入切好的美术资源

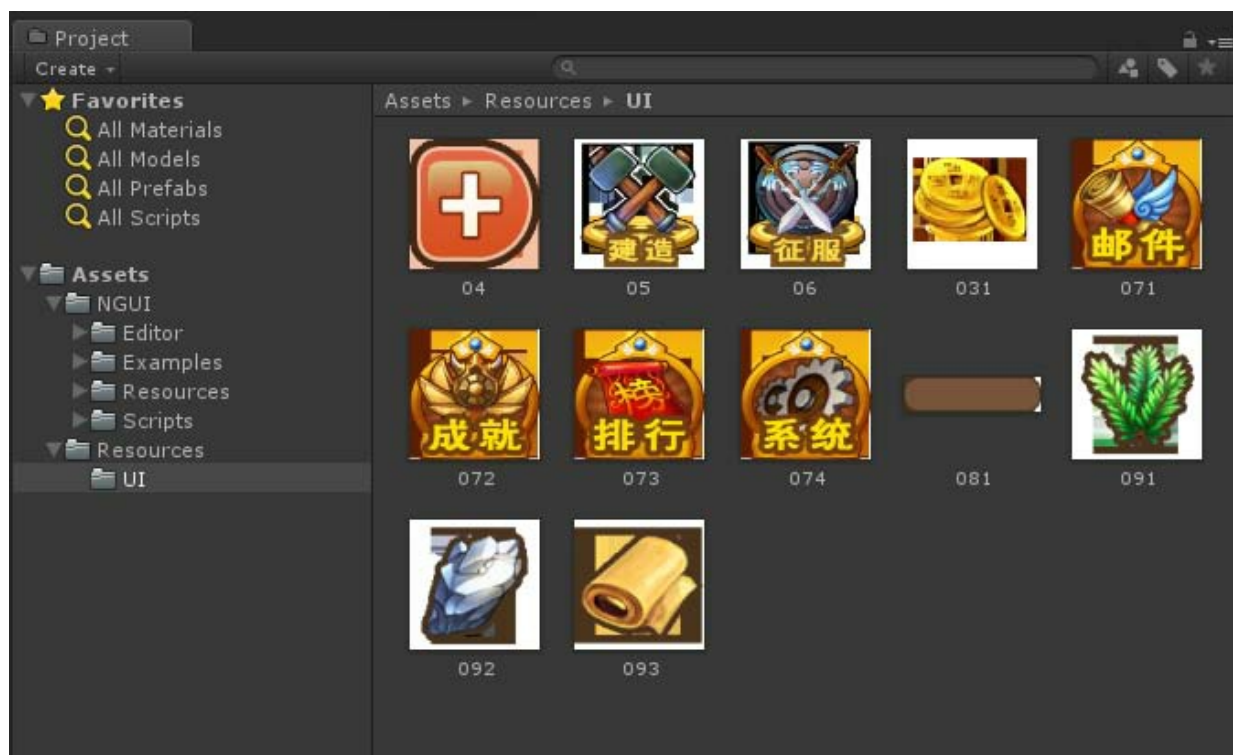
当美术人员提交给你大量零碎的 UI元件时，就可以开始进行UI的制作了。制作的首要任务是先将资源导入到引擎中去。首先在Unity的

Project窗口下的Assets文件夹下面建立一个文件夹，将该文件夹命名为Resources，表示这个项目的资源都放在这个文件夹下面，UI资源也不例外。这个文件夹名字一定要设定为Resources，不能改动。

这里讲一下为什么一定要设定一个名为Resources的专门的资源文件夹。Unity开发中，如果涉及动态加载（在游戏中触发了某个条件才需要加载）的情况，比如需要临时生成一个烟火特效等，都会用到Unity的资源加载方法：Resources.Load（）；这要求需要被动态加载的资源一定要放在Assets下面一个叫Resources的文件夹中。而UI经常会涉及根据不同情况动态加载UI模块的情况，所以一般情况下，我们都会将UI的资源放在Assets目录下的Resources文件夹中。

但是也不是所有资源都放在Resources文件夹中就万事无忧了。因为Unity在生成游戏安装包时，对于Resources以外的文件夹，只会打包场景中用到的资源文件，而对于Resources文件夹，因为涉及动态地往内存里加载资源，所以Unity会无条件的全部打包。如果不加考量的把所有资源都放到Resources目录下，可能会导致最终得到的游戏安装包体积变大。

这时我们可以在Resources目录下建一个名为UI的文件夹，用来单独存储UI资源，以避免UI资源和其他的角色、特效等资源放在一起难以管理。全部选中美术人员提交的UI元件，一起拖曳到Unity界面中的“Resources\UI”目录下，等待Unity读取资源之后，我们导入的资源会显示在“Resources\UI”目录下，这样就完成了资源的导入，如图2.18所示。



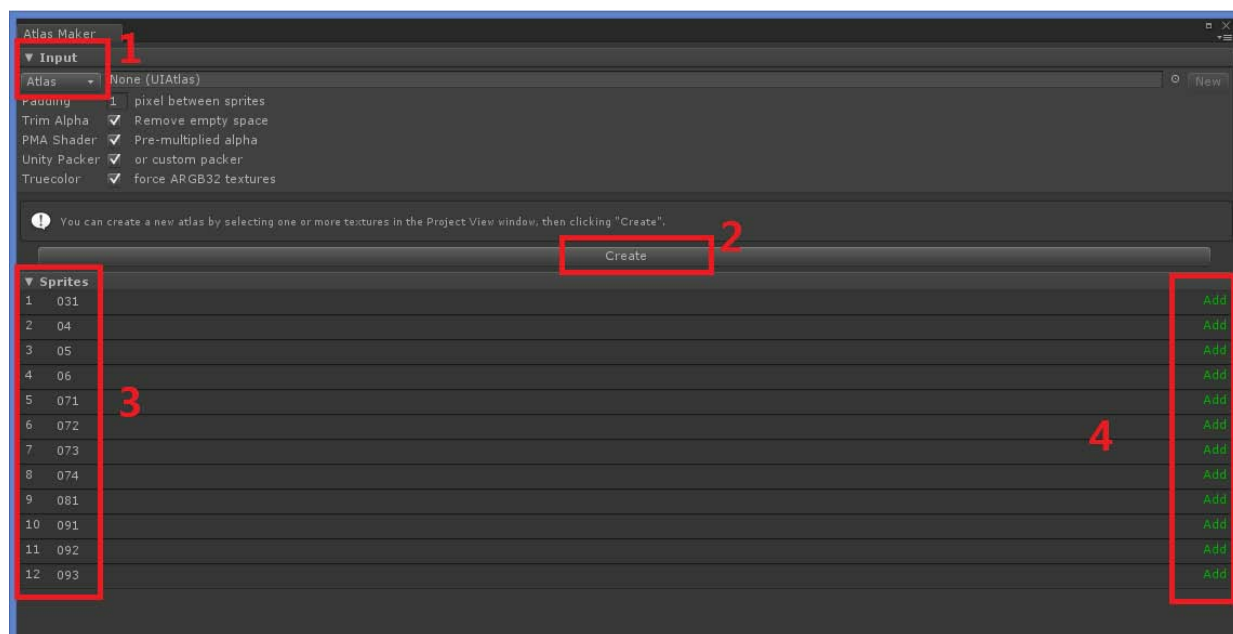
▲ 图2.18

### 2.3.3 用Atlas Maker制作图集

刚刚我们已经将 UI 元件的源文件全部导入到了 Unity 中相应的文件夹目录下，在 Unity 的 Project 窗口中全部选中刚才导入的 UI 元件，单击鼠标右键，选择最顶部 NGUI 菜单，选择 Open Atlas Maker（Atlas Maker 是 NGUI 自带的一个 UI 图片打包工具），这样就能自动将这些 UI 元件放入到 Atlas Maker 中，如图 2.19 所示。

在 Mac 系统下，右键菜单如果没有 NGUI 选项，那么可以选中导入的 UI 元件，在 Unity 顶部的菜单栏中，选择 NGUI 菜单，然后选择 Open\Atlas Maker 即可。

打开后的 Atlas Maker 界面如图 2.19 所示。



▲ 图2.19

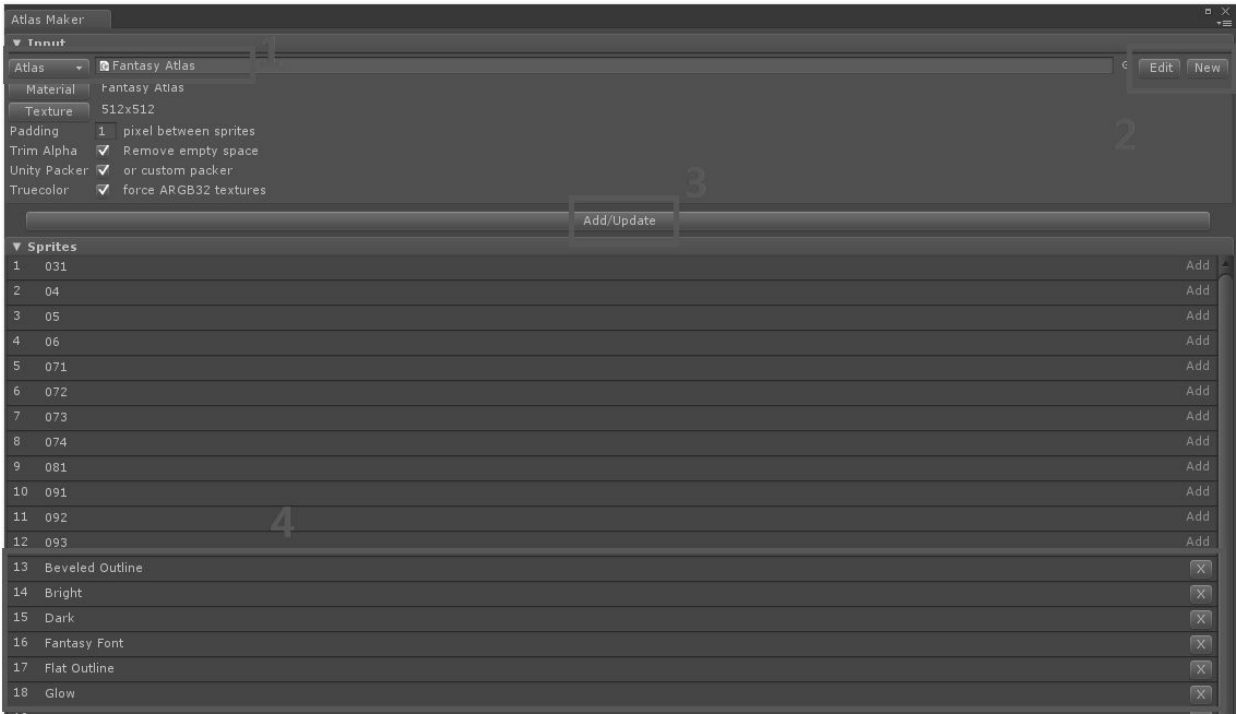
图2.19中，标号为1的红框是已有图集的选择按钮，如果你需要将这些新导入的UI素材资源全部新增到一个已有的图集里，就可以单击这里。单击后能看到当前项目工程中已有的所有的图集，然后可以选择其中一个图集，此时标号为2的红框处的按钮将变成 **Add/Update**，这样就可以新增或者更新这批资源到已有的选定的图集中了。

图2.19中标号为2的红框是主按钮，当要打包的UI素材资源没有选定打包到某个已有图集中去时，这个主按钮会显示**Create**，意为用这些资源创建一个全新的图集。如果通过标号1的红框处的按钮选择了一个已有图集的话，这个主按钮将变成 **Add/Update**，意为新增/更新当前这批UI资源到选中的图集中。更新的机制为同名的**Sprite**图片将会被替换。

图2.19中标号3处的红框显示的是当前选中的UI图片资源的序号和文件名称，标号4处的红框显示的是这些资源哪些是新增的；哪些是更新的；哪些是已有的。在这里，如果选中了一个已有图集，那么该图集的**Sprite**也会一起显示出来，如图2.20所示。在图2.20中，红框1则表示当前选中了一个已有的名为**FantasyAtlas**的图集。红框2是编辑图

集和新建图集的按钮。红框3的按钮由Create变为了Add/Update。红框4显示出了该图集已有的Sprite，在最尾部有一个删除按钮，单击之后，将会从这个图集中删除该Sprite。如果此时单击Add/Update按钮，那么Sprites列表中尾部标记为绿色的Add的精灵图片将会被新增到这个图集之中。

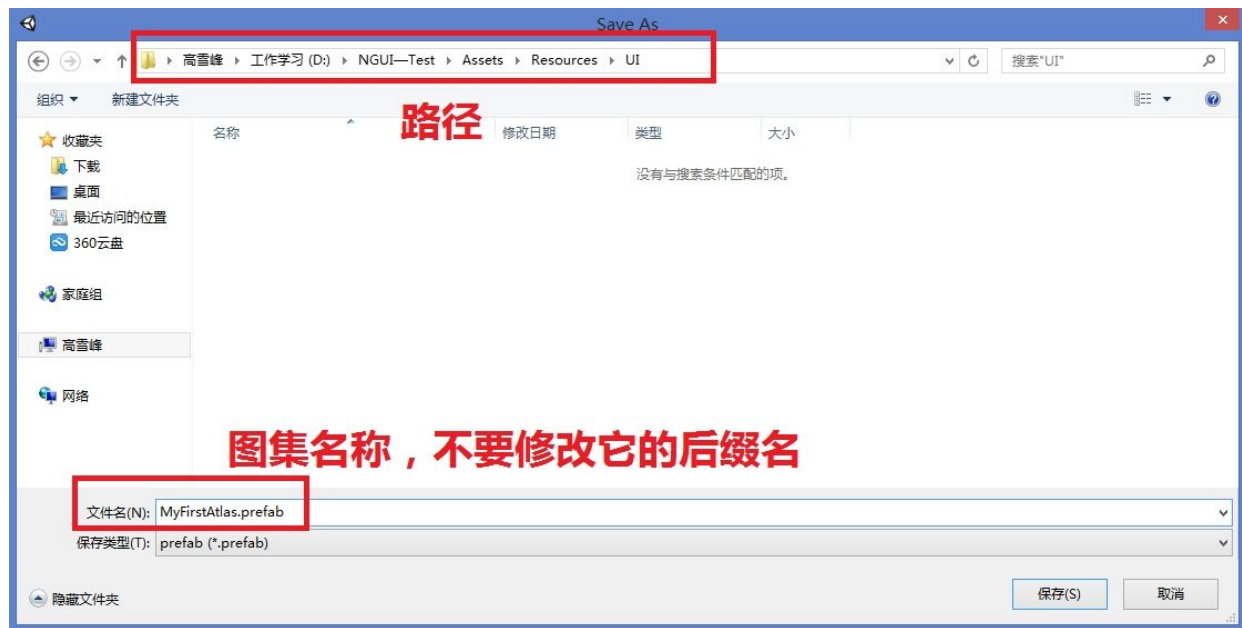
如果需要更新现有图集集中的某一个精灵，则将新的精灵图片文件的名称设为和它要替换的精灵的名称一样，然后按照以上步骤选择它要替换的精灵所在的图集，单击Add/Update即可实现直接替换资源。这在项目开发中是非常实用的，当美术人员希望修改UI资源、用更新的UI资源替换之前的旧资源时，这个自动更新功能将会让程序员非常方便。



▲ 图2.20

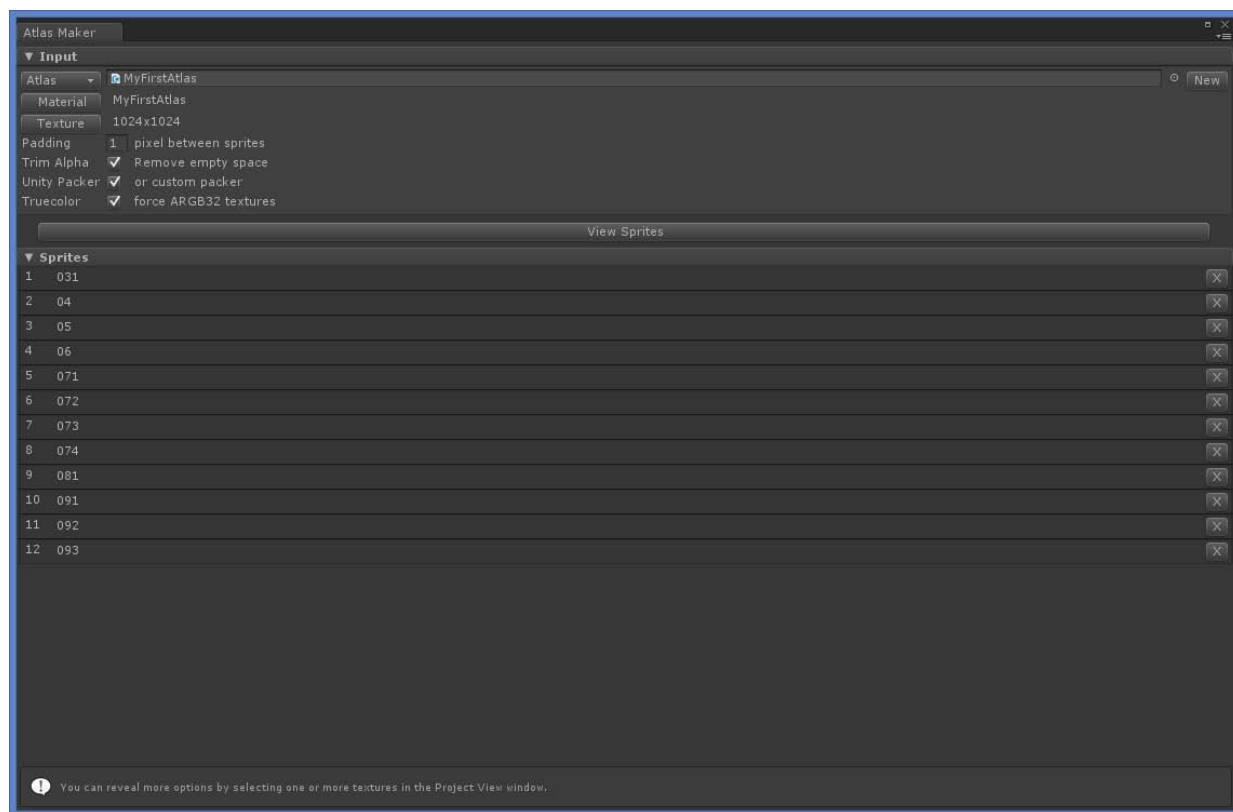
下面继续创建属于我们自己的第一个图集，当打开Atlas Maker之后，我们看到的是图2.19所示的界面。我们需要创建一个全新的图集，所以单击Create主按钮，然后会弹出Save As对话框，将路径定位到

Resources\UI目录下，然后将图集的名称改为“MyFirstAtlas”，单击“保存”按钮即可，如图2.21所示。注意，不要改变文件的后缀名，文件保存后是一个Prefab。



▲ 图2.21

单击“保存”按钮之后，我们可以看到Atlas Maker界面已经变成了如图2.22所示的情况，这表明图集已经创建成功，中间的主按钮变成了View Sprites，单击后可以预览该图集中所拥有的精灵。



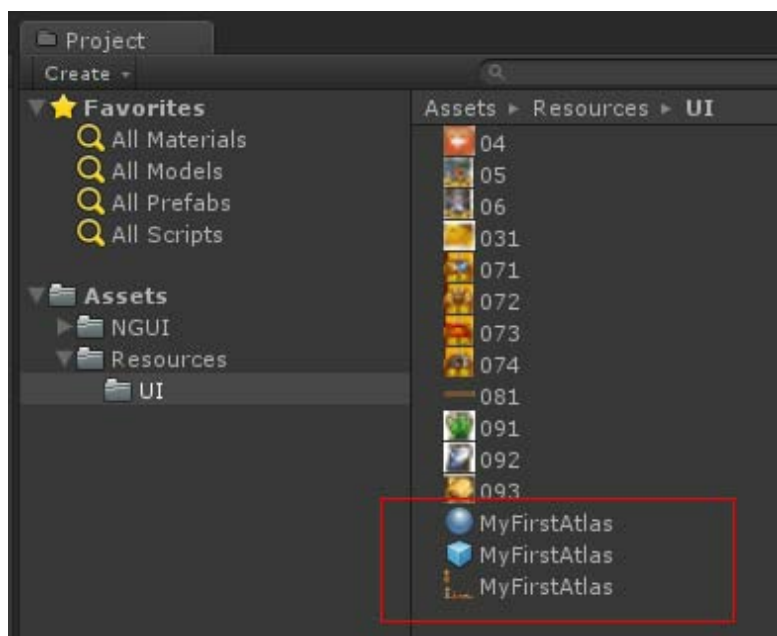
▲ 图2.22

关闭 Atlas Maker 界面，然后注意看 Project 窗口中，Resources\UI 目录下除了我们之前导入的 UI 图片资源以外，多出了 3 个名为“**MyFirstAtlas**”的文件，如图 2.23 所示。这 3 个文件是一个图集必须具备的 3 个文件：图集的贴图、图集的材质球和图集的预设体。其中，第一个球形图标的 **MyFirstAtlas** 文件为图集的材质球；第二个蓝色方块的文件为图集的预设体；第三个图片缩略图文件则是图集的贴图，也就是精灵合成为一张整图之后的图片。

至此，我们的第一个图集就算制作完成了，这个图集在后面制作 UI 时就可以直接使用了。针对 Atlas，还有很多功能和用法，例如九宫格等，我们会在本书的后半部分讲到。

特别注明：在制作完 UI 图集之后，我们可以将之前导入 Unity 的 UI 资源源文件删除以减小资源量。





▲ 图2.23

## 2.4 制作第一个UI字体

### 2.4.1 为什么要制作UI字体

在游戏的项目开发中，字体是经常会用到的东西，因为游戏中不论是聊天、公告、提示语还是界面显示，都会涉及用程序来写字。一般来说，会有系统默认字体供我们使用，但是出于以下两个原因我们经常会需要制作独特的字体。

- 系统字体的风格和美观程度等无法满足我们的需求。

一般来说，系统字体都比较死板、生硬，风格单一，经常无法满足项目需求。例如，我们希望游戏中所有文字都使用楷书来突出中国风，那么则需要我们自己植入楷书字体。再例如，我们需要在某些地方显示一些造型独特的字体，更需要制作我们自己独特的字体文件才能满足这种需求。

- 应对系统字体丢失的情况。

某些玩家（特别是安卓玩家）如果经常从网上下载一些个性化的字体管理软件，会很容易导致系统字体丢失，这种情况一旦发生，会导致游戏内所有的文字都不能正常显示。为了以防万一，我们需要植入一套自己的字体在游戏资源包内部。

### 2.4.2 静态字体和动态字体

我们在2.2节中已经介绍了什么是静态字体和什么是动态字体，这里我们来了解一下什么情况下需要静态字体，什么情况下需要动态字体。

当有大面积的字体需求，并且需要的文字几乎涵盖大部分汉字时，我们就需要制作动态字体。与其说是制作动态字体，不如说是植入动态字体，因为在新版的NGUI中，制作动态字体只需要导入一个TTF格式的字体文件即可。

当在某些地方有特殊字体的需求，并且这种字体显示的文字有限时，例如受到伤害时，角色头顶需要飘出一个有艺术效果的数字来表示伤害量，这种字体效果只会显示0~9共10个数字而已，其他地方都用不到这种字体，那么这个时候我们就可以为它制作一个静态字体。

具体来说，静态字体和动态字体有以下实质区别。

- 静态字体中，如果需要用到的文字不多，打成图集后资源量往往比动态字体小，一个动态字体的TTF格式文件一般为3~6MB。

- 静态字体可以通过提供一张自定义的含有所需文字的图片和一个配置文件（记录图片哪一块是哪个字的文件）来完成。动态字体只能通过导入整个TTF格式的字体文件完成。

- 静态字体中的字一般非常有限，只有极少数的字（否则图片资源会非常大），所以应用范围非常小，几乎很少用到静态字体。而动态

字体几乎包含所有的文字，被广泛运用。

### 2.4.3 制作静态字体介绍

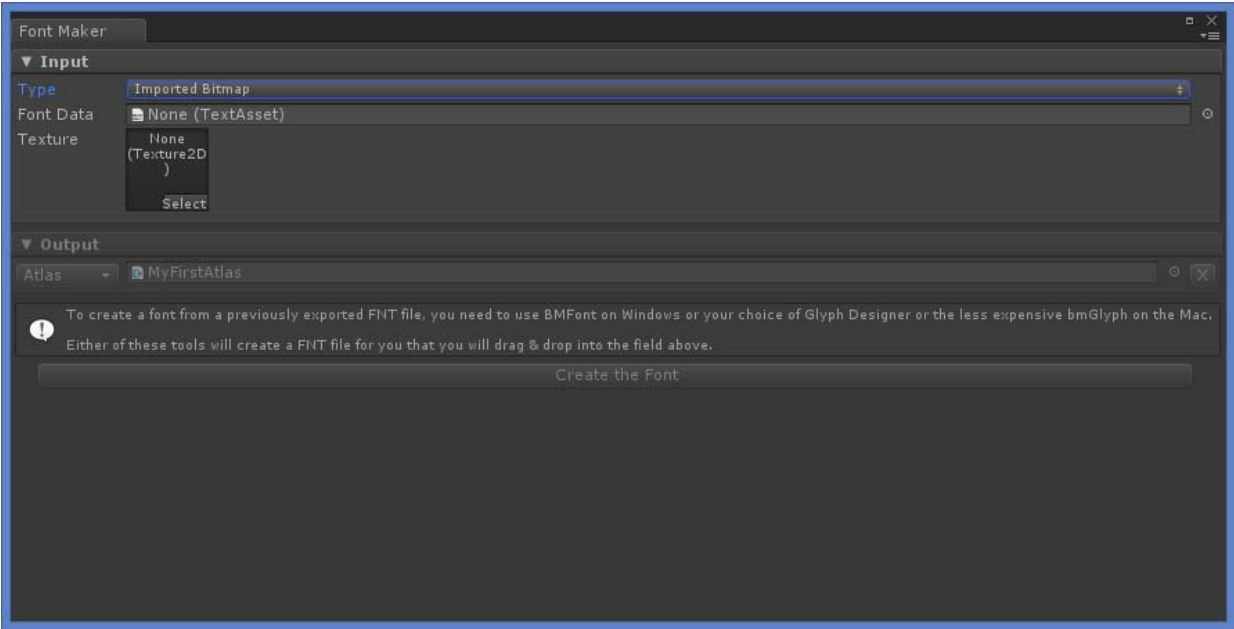
静态字体曾经风靡一时，原因是那时候NGUI旧版本对动态字体支持不是很好，所以很多时候得依赖静态字体。目前NGUI对动态字体支持很好，所以静态字体的应用就变得很少，只有在特殊情况下才使用。

要制作静态字体，需要将字筛选出来打成一个图集，并生成一份记录其中哪一块是哪个字的配置文件，这时可以借助一个名为BMFont的软件将其制作出来。制作出这两份文件之后，导入到Unity里。

在Unity界面中，在Project窗口内单击鼠标右键，选择NGUI菜单，选择Open BipMap Font Maker，打开流程和打开Atlas Maker极其相似。Mac电脑可以通过Unity顶部菜单栏中的NGUI菜单打开。

然后会弹出如图2.24所示的界面，在Type 中选择Imported Bitmap，然后在Font Data中选择我们之前制作出的那个记录文字位置信息的配置文件（最好是TXT格式），在Texture中选择我们之前制作出的那个文字图集，然后单击主按钮Create the Font，即可创建一个静态字体，创建出来的字体文件和制作图集得到的文件类似。制作好后，以后我们需要用字体的时候，选择这个字体的预设即可。

注意，制作完成后不要删除导入的那张文字图集和文字位置的配置文件的源文件，如果删除将会导致字体读不出文字。



▲ 图2.24

## 2.4.4 制作动态字体介绍

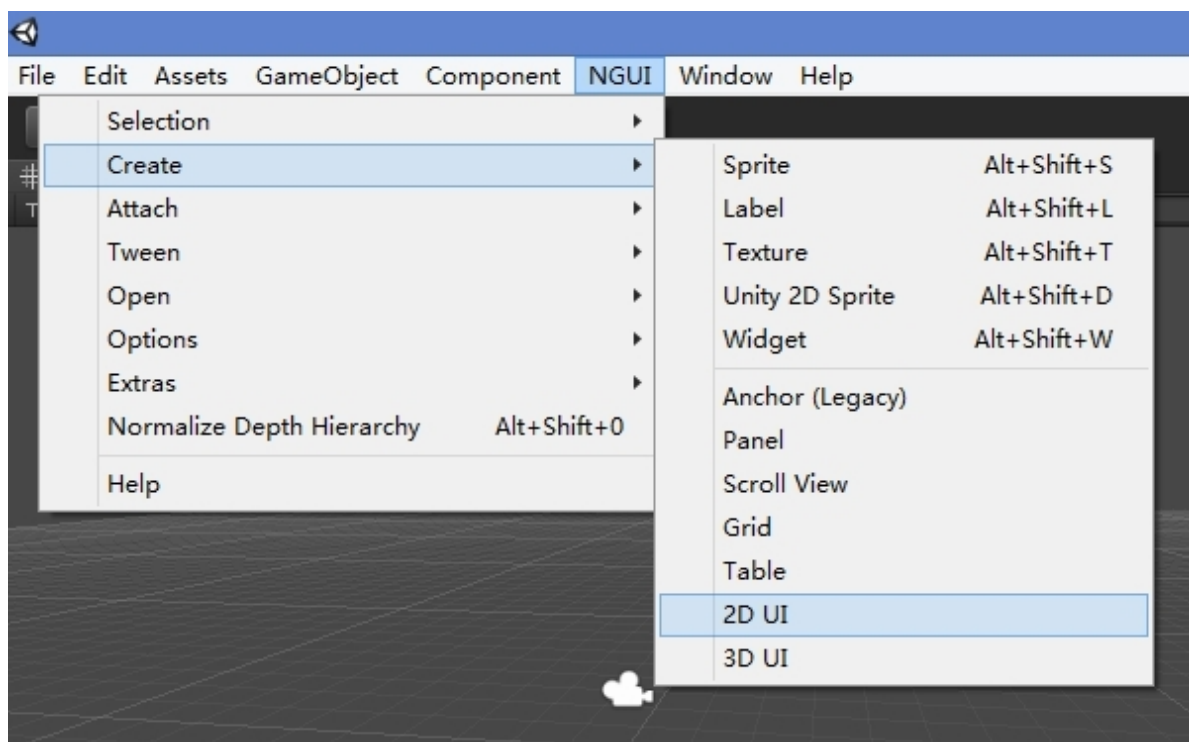
在新版本的NGUI中（比如，3.6以后），制作动态字体非常简单，只需要从网上下载一个TTF格式的字体文件即可。然后将这个字体文件导入Unity中就算完成了，以后需要用字体的时候，就能直接调用这个字体。

注意，字体文件要选择简体中文的，TTF文件大小一般为3～6MB，如果远远超出了这个数，一般来说很有可能是字体中包含了很多种语言。

## 2.5 创建第一个UI

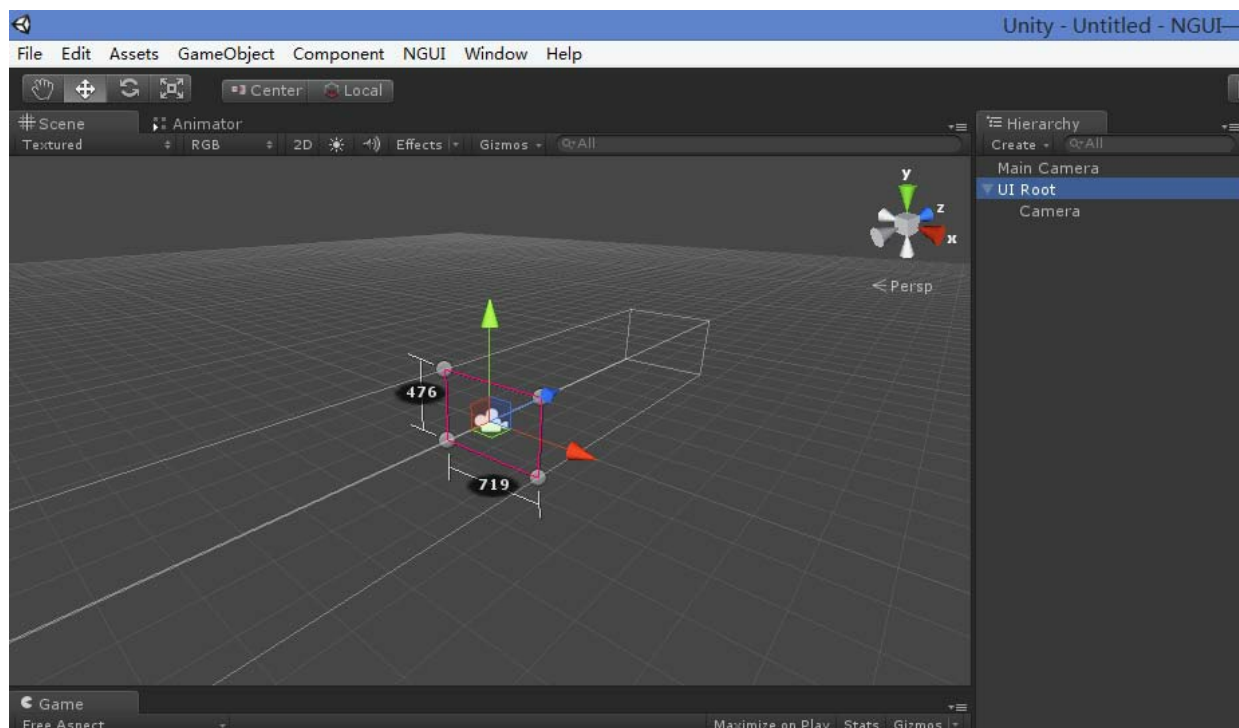
### 2.5.1 创建一个2D UI

制作UI时，首先我们要创建UI的“根”。在Unity顶部NGUI菜单中选择Create，然后选择2D UI，如图2.25 所示。



▲ 图2.25

创建完成后，我们能看到图2.26所示的景象，在Scene窗口中，NGUI自动生成了一个名为UI Root 的物体，其中带有一个Camera 作为子物体。



▲ 图2.26

这个新生成的Camera，是NGUI生成的专门用来渲染UI的相机，当我们生成NGUI的UI Root时，就自动将生成的UI的layer设为了第8层。在这个相机中，只能看见第8层的物体，也就是只能看见UI。因为是2D UI，所以我们可以从图中可以看到相机是正交相机。

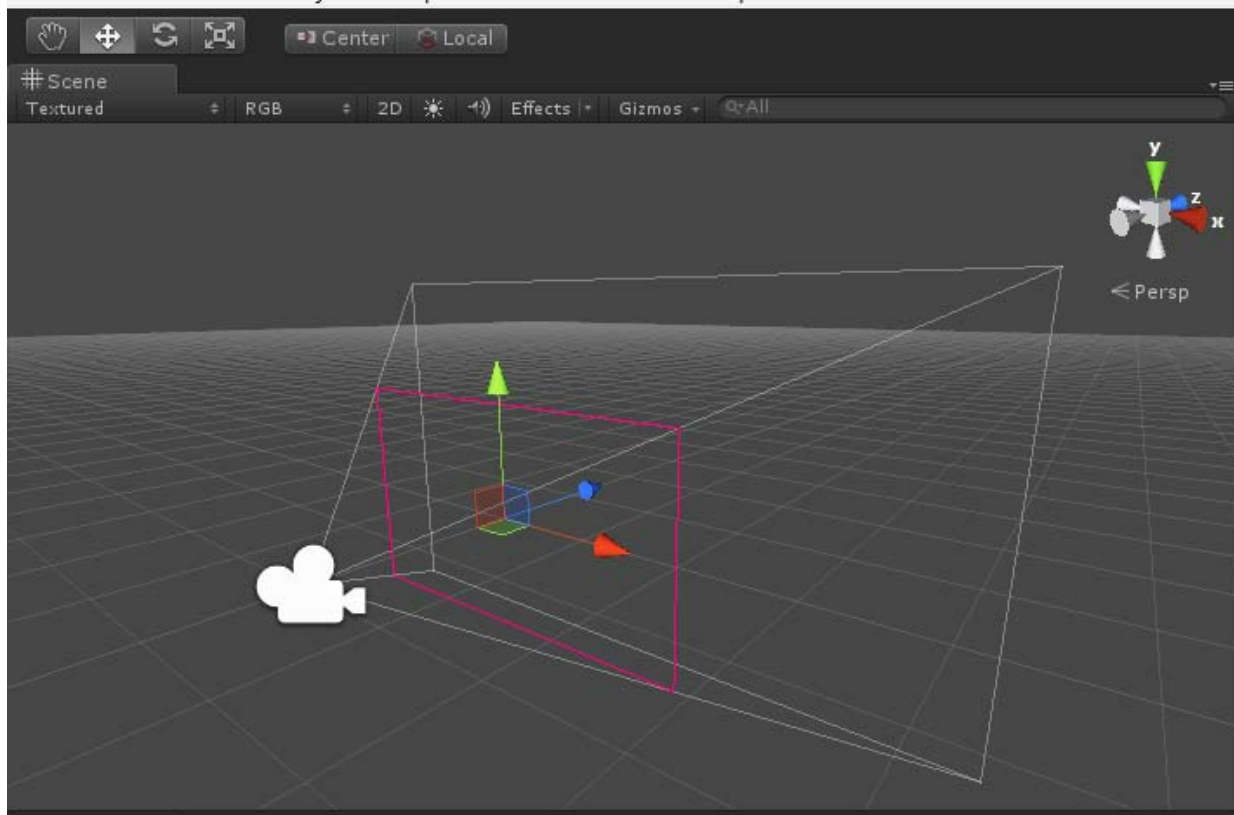
图2.26中红色的矩形是相机的视窗大小比例，它会根据Game视图中的屏幕长宽比设置自动调整。

### 2.5.2 创建一个3D UI

创建3D UI的过程和创建2D UI的过程类似，创建出来的3D UI如图2.27所示，NGUI依然自动生成了一个名为UI Root的物体，并带有一个Camera子物体，这个原理和2D UI类似，其中最大的区别就是相机的模式。3D UI的相机在Scene视窗中是一个正交摄像机，将会支持UI的三维透视效果。

### 2.5.3 了解UIRoot、UIPanel和UICamera组件

在我们创建的UI中，可以发现UI Root物体和Camera 物体上面都带有NGUI特有的脚本组件，其中 UI Root 物体上带有 UIRoot 和 UIPanel 两个组件，而子物体 Camera 带有一个UICamera组件，这几个组件都是NGUI体系中比较核心的组件，下面我们来简单了解一下。



▲ 图2.27

#### 1. UIRoot组件

UIRoot组件总是出现在NGUI的UI“树”的最顶层，也就是那个“根”物体中。它的作用是缩放UI。我们在让美术人员作图时知道，UI一般都是以像素作为单位，比如1920\*1080等，而Unity中则是以米为单位，如果一个100\*100像素的UI元件放入到一块1000\*1000分辨率的屏幕中，按理说这个UI元件应该是屏幕大小的1%，但是因为Unity中的单位是米，所以它会从100\*100像素的大小变为100\*100米，会导致一个小



UI元件变得非常之巨大。这个时候UIRoot会通过屏幕来缩放UI控件，让UI控件从视觉上是正常的。

在UIRoot组件中，它提供了3种缩放的方式，也就是UIRoot组件下的Type值。这3种方式分别为PixelPerfect、FixedSize、FixedSizeOnMobiles。

PixelPerfect是指永远保持像素大小不变，比如一张100\*100像素的图片，在500\*500分辨率的屏幕上，它是100\*100像素，在1000\*1000分辨率的屏幕上，它依然还是100\*100像素，因为它的源文件就是这个大小，而PixelPerfect让它一直保持这个大小。这样就可以让UI的图片永远是最清晰的，但是会导致分辨率高的屏幕下UI显得特别小；分辨率低的屏幕下UI显得特别大。

FixedSize是和上一种缩放方案完全相反的方案。在FixedSize下，NGUI将不再保护图片的原尺寸，只会关心NGUI自己所需要的缩放参数，这种模式下必须设置UIRoot的ManualHeight值，然后NGUI会将所有的控件按照和这个值的高度比例进行缩放。例如，设置ManualHeight为1000，然后一张100\*100的图片在高度为1000的屏幕分辨率下占1/10的高度，那么当UI放到一个分辨率为500\*500的屏幕上时，它依然占1/10的高度，只不过图片尺寸被自动放缩为50\*50，这样就保证了UI和屏幕分辨率的比例是一定的。

FixedSizeOnMobiles是两种方案的结合体，它会让UI在PC、Mac、Linux系统下自动采用PixelPerfect，而在移动设备上自动采用FixedSize。

如果没有选择FixedSize，那么必须设置另外两种缩放模式下的MinimumHeight和MaximumHeight两个值，代表最大高度和最小高度。例如选择PixelPerfect模式，将MinimumHeight设置为720，将MaximumHeight设置为900，那么在一个分辨率为800\*600的屏幕上，因为屏幕分辨率的高度小于UIRoot里的最小高度，UIRoot就会按照

FixedSize模式下ManualHeight为720的情况进行处理；同理，如果将UI放到一个分辨率为1920\*1080的屏幕上，因为该屏幕分辨率的高度1080大于设置的900，于是UIRoot就会按照FixedSize模式下ManualHeight为900的情况进行处理。

值得注意的是，在3.7.0以后的NGUI上，UIRoot的缩放模式改为了。

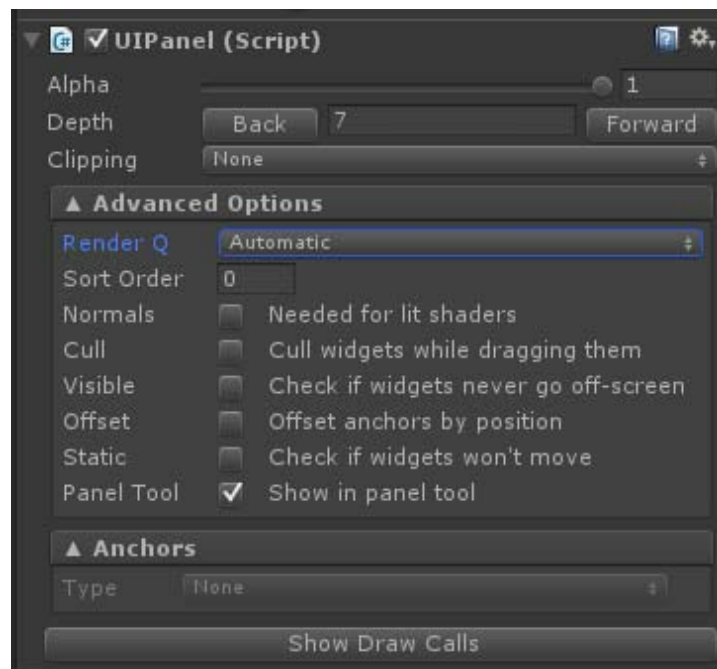
- Flexible，等同于上文讲到的PixelPerfect。
- Constrained，等同于上文讲到的FixedSize。
- ConstrainedOnMobiles，等同于上文讲到的FixedSizeOnMobiles。功能上几乎完全一样。

## 2. UIPanel组件

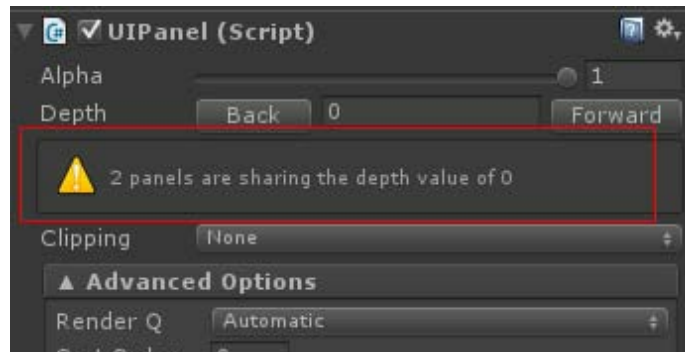
如图2.28所示，UIPanel有很多属性。其中，Alpha属性顾名思义是透明度，默认为1不透明。它将控制它旗下所有Widget（所有的UI控件都将带有Widget，因为它们都继承自Widget）的透明度，也就是它会让它的子物体里的所有UI控件都一起发生透明度变化，可以用来做整个UI的淡入淡出以及隐藏等。

Depth深度属性，这是一个极其重要的属性。在NGUI中，每一个Panel都有Depth，每一个Widget控件也有Depth，Depth将决定渲染的顺序，直接影响了UI之间的前后重叠关系。Depth越高的控件将会显示在视野的上层，Depth越高的Panel也会显示在视野的上层。但是Panel的Depth权重远远高于Widget，也就是说，在大部分情况下，属于低Depth的Panel的控件，不管这个控件本身的Depth为多少，它都将显示在高Depth的Panel的控件后面。当你有多个Panel的时候，例如你制作了很多面板界面，每一个界面都有一个Panel，那么此时尽量保证这些Panel不要共用同一个Depth，因为这将导致NGUI在渲染时无法以1个DrawCall完成，会以增加DrawCall的方式来保证渲染顺序不混乱，这样

就增大了性能的开销。不过NGUI在碰到Panel有共用Depth的情况时会做出提醒，如图2.29红框部分所示。



▲ 图2.28



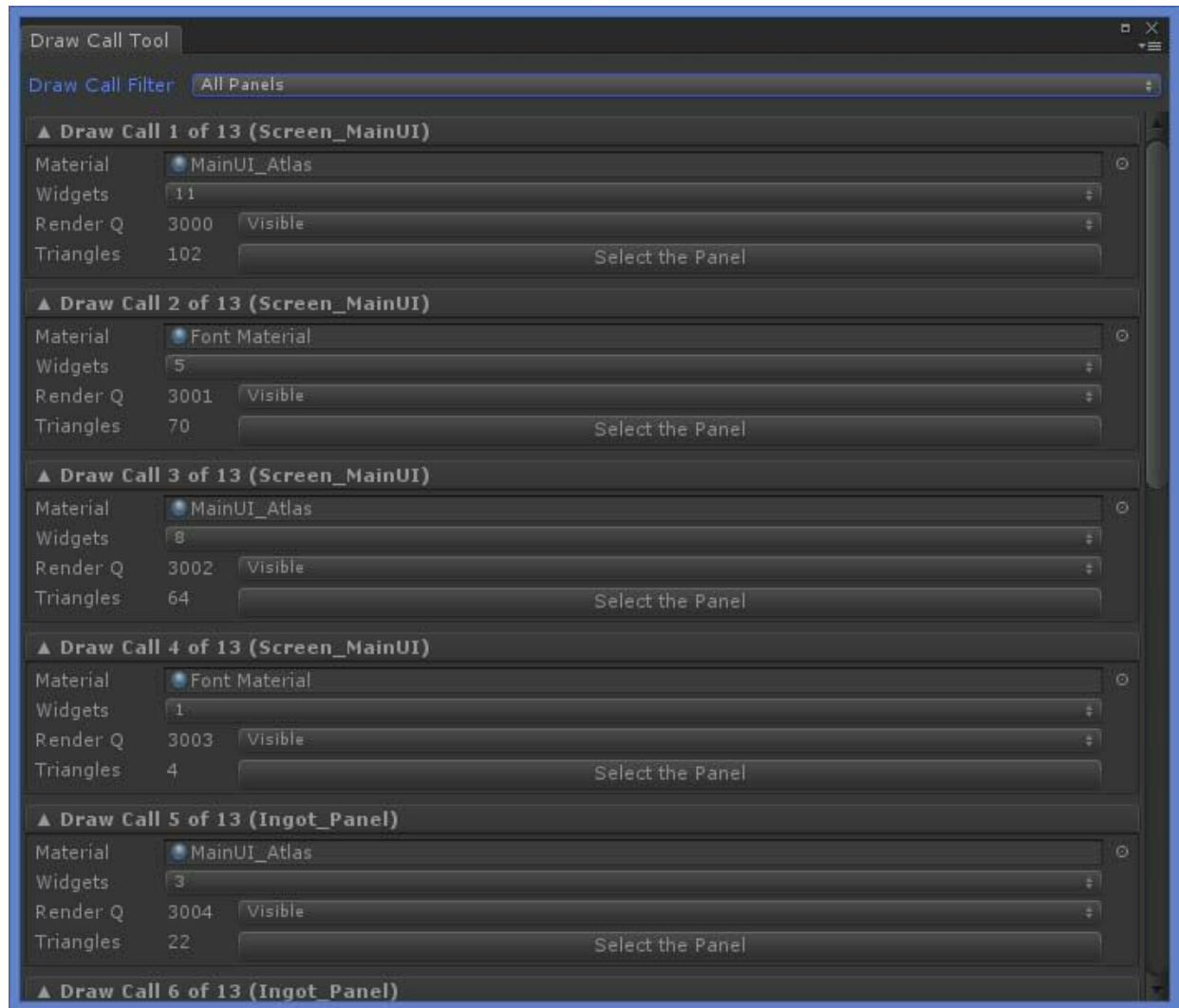
▲ 图2.29

Clipping是剪辑视窗的功能，它将可以让一个面板只显示某一块区域，关于这部分知识后文我们再讲解。

在高级选项中，我们讲解一些初学者需要了解的。Render Q可以理解为渲染顺序，默认为自动设置。这个选项在和粒子系统结合使用时会有影响，我们后文会说明。如果该Panel下的UI需要被灯光影响到（NGUI的UI是默认不接收灯光照射效果的），需要勾选Normals。如

果该Panel下面所有的UI控件都不会被移动，那么可以勾选Static来将它们设置为静态的，这样会导致该Panel下所有的控件都将忽略位置、旋转、缩放的操作，永远保持不动。虽然这样可以提高一些性能，但是慎重使用。

单击Show Draw Calls按钮，可以看到该Panel下所有的DrawCall消耗情况，如图2.30所示。

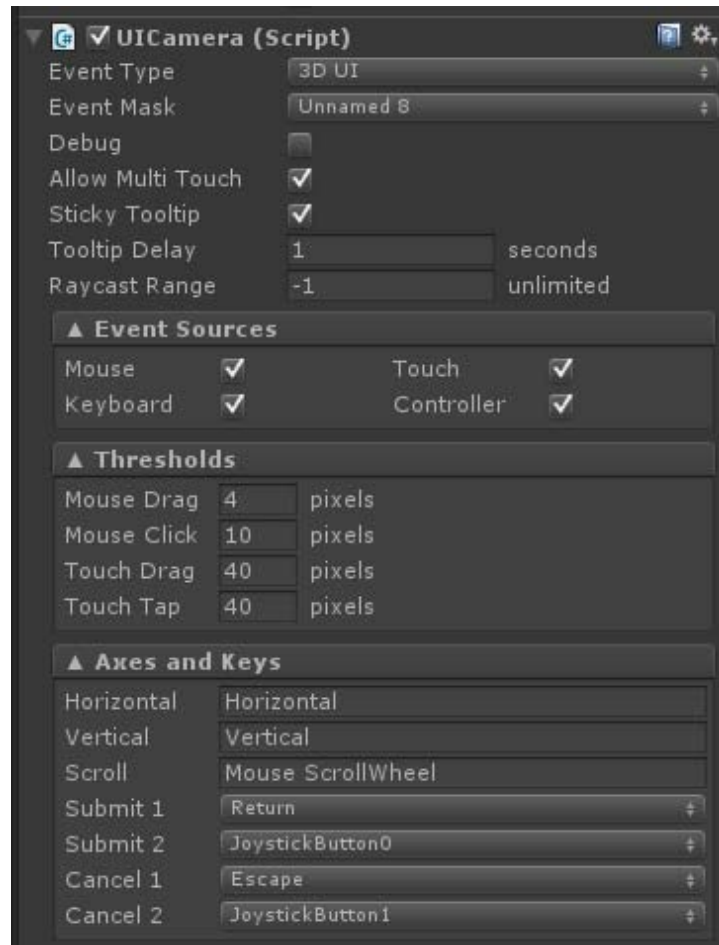


▲ 图2.30

### 3. UICamera组件

图2.31所示为UICamera组件的截图，UICamera这个组件的核心作用是：让带有这个组件的摄像机渲染出的物体能够接收NGUI的事件。

如果我们自己创建了一个物体，并且希望对这个物体使用一些 NGUI 中的事件，例如 `OnPress()`、`OnDrag()`等，就需要为渲染这个物体的摄像机添加UICamera组件。



▲ 图2.31

在UICamera中，大部分设置我们都不需要去操心，它让我们的事件支持多点触摸、鼠标键盘触摸屏等事件的接收。但是要注意的是 **EventMask** 这个选项，这个 **EventMask** 和相机中的 **CullingMask** 非常相似，相机的 **CullingMask** 是为了选择渲染那些层的物体，这里的 **EventMask** 是为了选择接收那些层的物体的事件。UICamera会默认只接收我们创建UI时被自动设置的那个layer，但是，如果我们在制作UI过程中，在创建UI后因为某些原因修改了UI的层，一定要将UICamera的

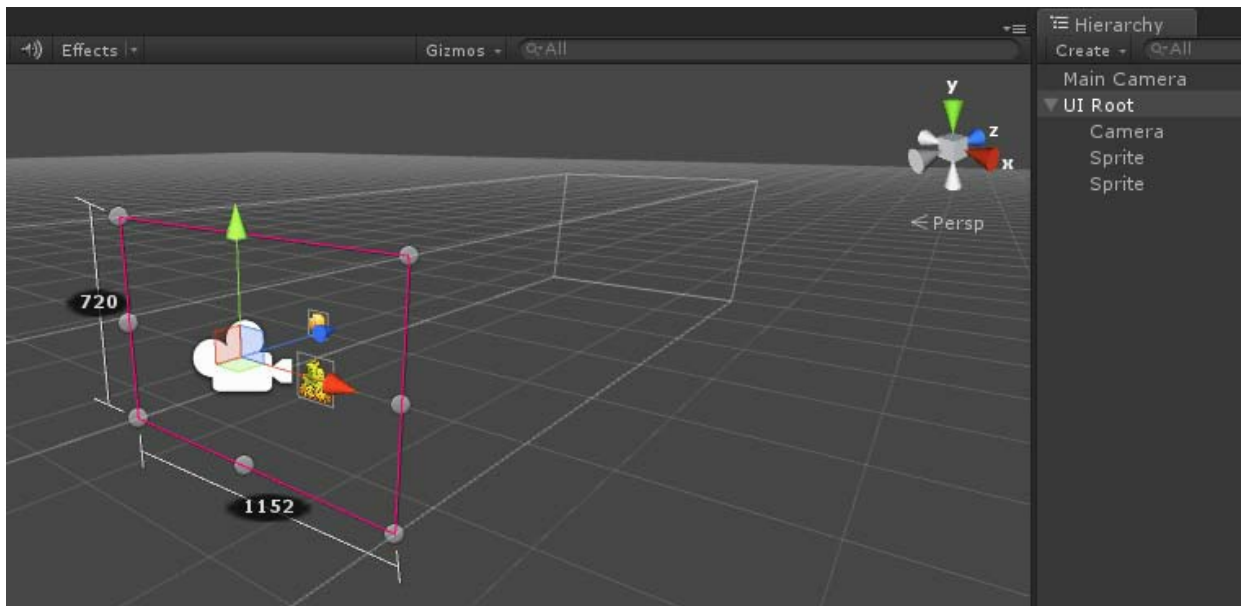
EventMask修改过来，否则将会发现，我们单击UI没有反应，因为它接收不到这个layer的物体事件。

关于这3个最基础的控件讲了这么多，其中有很多都是较少用到，主要目的是加快对NGUI概念的形成。具体在需要的时候应该进行什么样的操作，我们后面的一些实战内容中会讲到。

## 2.6 2DUI和3DUI的工作原理

### 2.6.1 2DUI的工作原理

先创建一个2DUI（创建方法上文已讲过），然后在2DUI的“根”UIRoot下创建两个精灵（创建方法后面会详解）。然后得到的效果如图2.32所示。

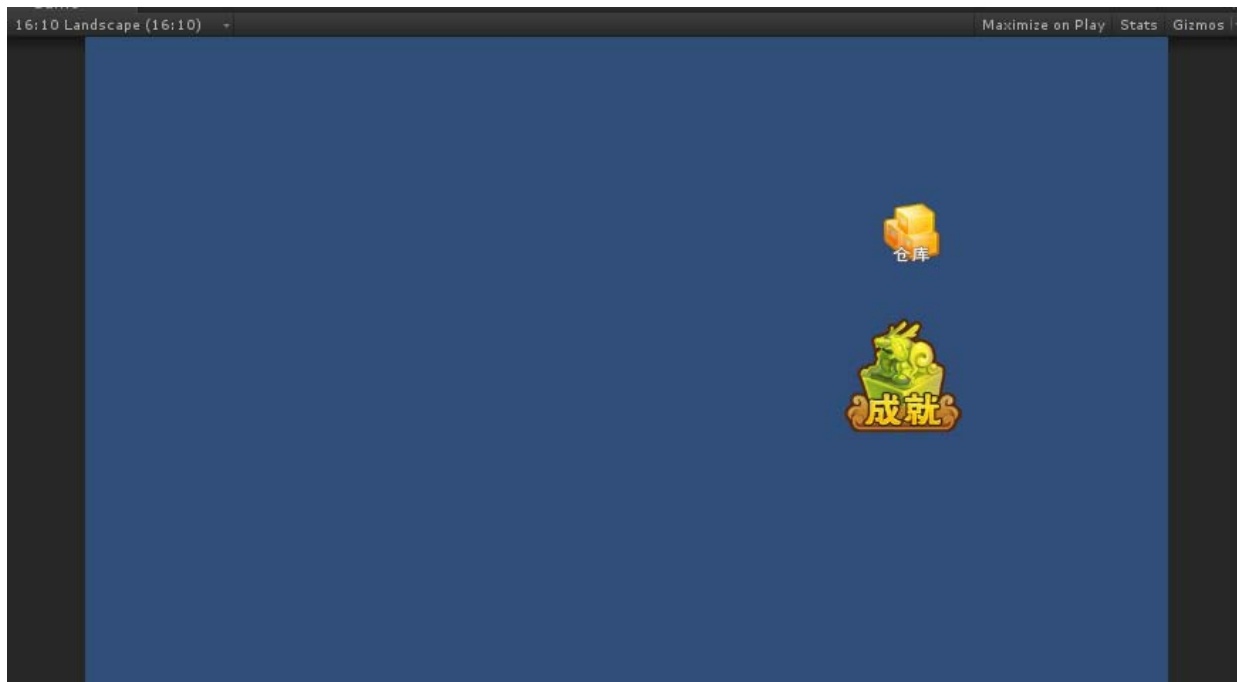


▲ 图2.32

从图2.32中可以看到，创建的两个Sprite为两个按钮图片，它们的位置在UIRoot的红框（视窗）上，也就是Sprite的z轴、相机的z轴、UIRoot的z轴都为0，因为2DUI都是纯粹的2D图片按层次显示，不会出

现三维立体效果，所以都是直接紧贴着视窗，只要UI控件在UIRoot的红框范围内，那么UI就能够正常显示在Game上。在Game视图中，我们看到的景象如图2.33所示。

2DUI最本质的意义是：UI摄像机是一个正交摄像机。

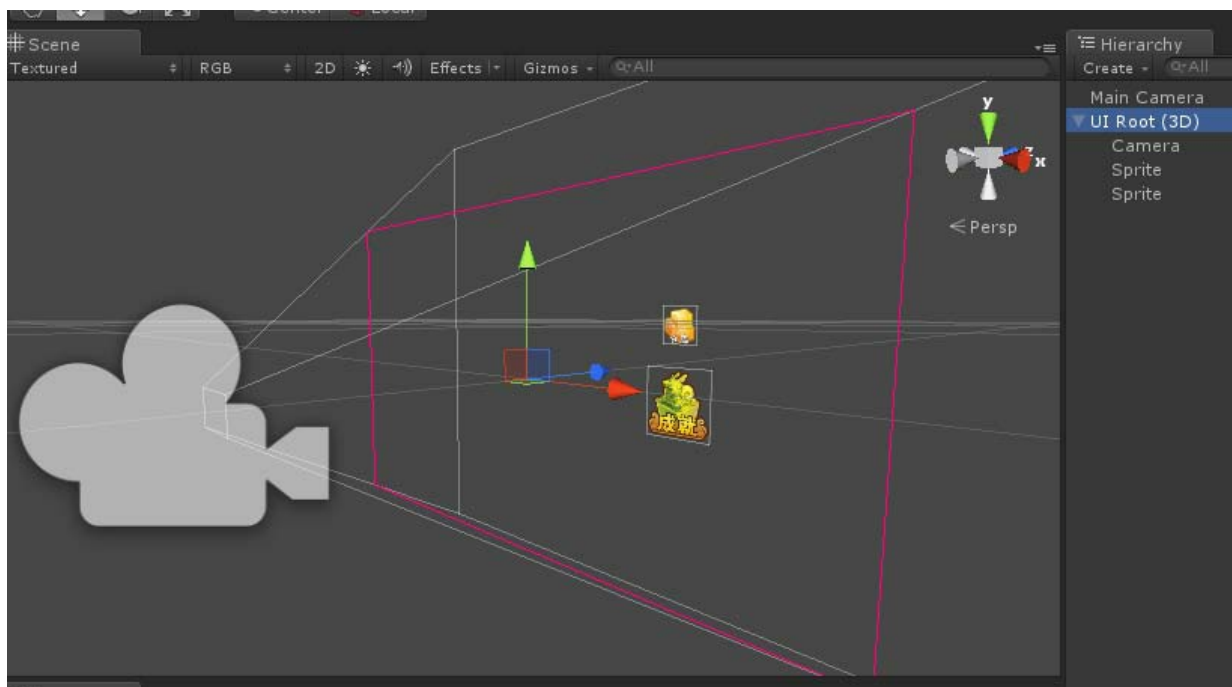


▲ 图2.33

### 2.6.2 3DUI的工作原理

同上2.6.1小节一样，先创建一个3DUI，然后在“根”下面创建两个Sprite，得到如图2.34所示的景象。





▲ 图2.34

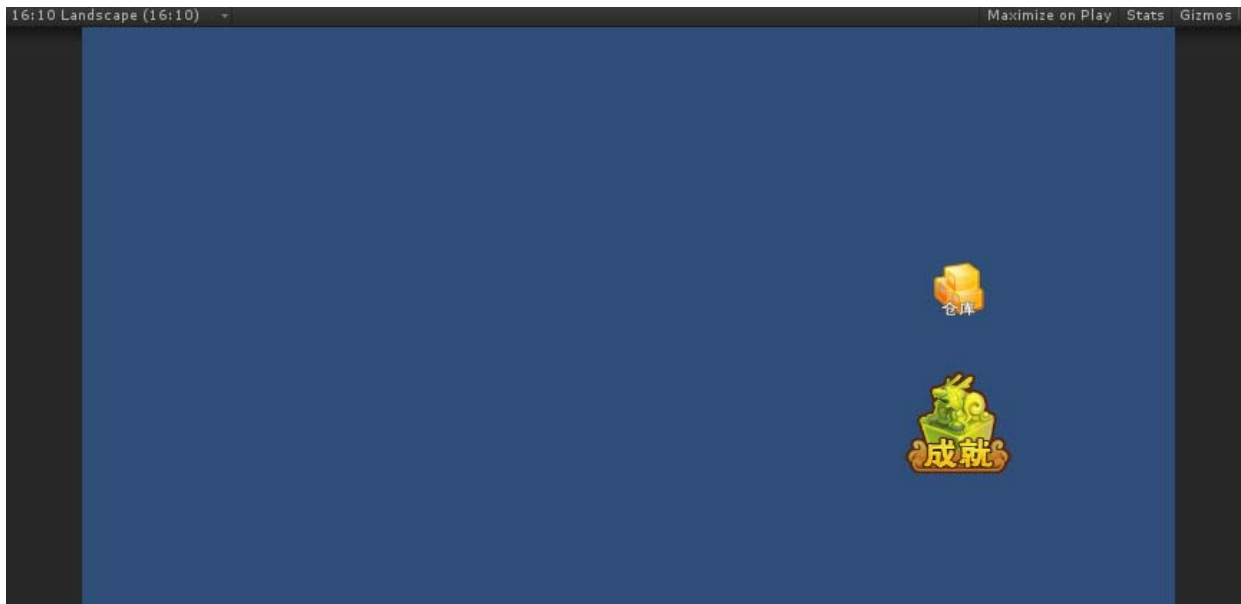
从图2.34中可以看到，在3DUI下，创建的UI控件都在一个三维立体空间中，摄像机是一个透视的摄像机，这和2DUI有着截然不同的区别，因为2DUI是一个正交摄像机。

3DUI中UIRoot的坐标点如图2.34所示，是在三维空间的一个点上，这个位置是创建UI时自动定义好的，以后创建的UI控件都会自动地在这个点所在的面上生成（自动统一到UIRoot的z轴，因为Sprite属于UIRoot的子物体），所以图2.34中UIRoot根物体和两个创建的Sprite的z轴都为0，而Camera的z轴为-700。

我们在图2.34生成的两个Sprite用3DUI做出的效果，在Game视图里看，和2DUI几乎是一模一样的，如图2.35所示。

但是需要注意的是，如果需要将2DUI改为3DUI，不是简单地将摄像机改为透视模式就行了，在NGUI 3.6.0以后的某些版本中，这样会导致看不到任何UI控件（这些版本的2DUI的控件和Camera、UIRoot三者在同一个z轴面上，而变成3D摄像机后是看不到和摄像机同z轴的物体），如果将3DUI的摄像机直接改为正交模式，也并不能简单地变成

2DUI，因为正交相机的Size并不和NGUI默认的值一样。所以，在进行UI开发时，最好先思考一下将要使用2D还是3DUI更好。



▲ 图2.35

### 2.6.3 如何判断该选择哪一种UI

上文中我们讲解了2DUI和3DUI的原理，那么在实际的项目开发中，如何来判断应该使用哪一种UI呢？有以下一些规律可以参考。

(1) 新版本的NGUI对3DUI支持很好，如果3DUI和2DUI选择哪一个都行的情况下，建议选择3DUI，扩展性更强。

(2) 如果出现 UI 不允许有远近透视的大小变化，必须选择2DUI。例如，角色头顶的血条，在人物跑远了之后如果血条并不会变小，这时血条就必须用2DUI做。

(3) 如果要出现UI有三维变换的效果，例如，由远及近的变大、三维旋转等，就必须用3DUI。

(4) 无法明确知道应该用哪一种UI的情况下，建议选用3DUI。

(5) 不论用哪一种UI，其实本质上只是一个摄像机的区别，基本上都能实现UI效果，只是需要的处理不一样，所以，如果选择错了UI

模式也不用太着急，总有很多办法来解决的。

## 2.7 深度（Depth）概念

### 2.7.1 强化对深度的理解

深度的概念将会一直伴随着UI的制作过程，是UI中一个非常重要的概念。我们在2.5.3小节中讲解UIPanel时已经讲解了深度的概念，这里我们再强化一下对深度的理解。在老版本的NGUI中，UI的显示层次关系是依靠z轴进行的。在新版本的NGUI中，所有UI的z轴都被统一，然后用深度来决定和管理显示的层次关系。关于深度，我们要记住一下关键点。

（1）每一个UIPanel和每一个UI控件都一定会有一个Depth，深度值大代表显示的优先级高（会越趋向于在界面更上层显示）。

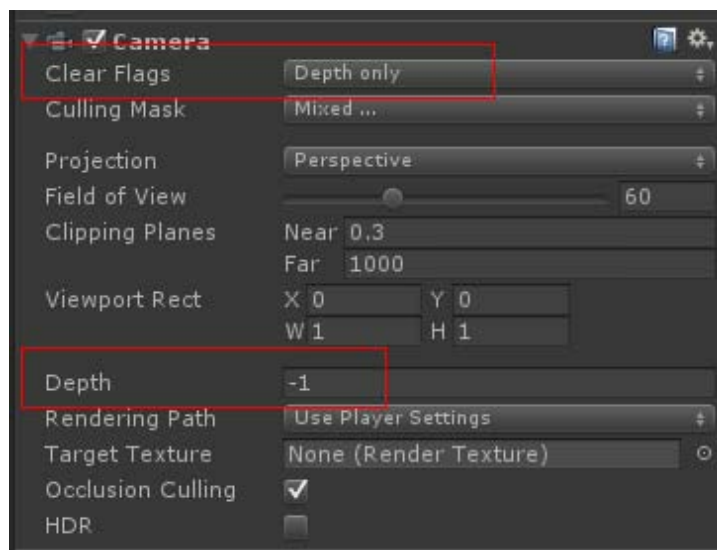
（2）Depth决定的是UI的显示层级关系，一个UI控件是否显示在最上层是由它所属的Panel的Depth和它本身的Depth决定的。一般情况下，属于低Depth的Panel的控件，不管这个控件本身的Depth为多少，它都将显示在高Depth的Panel的控件后面（被高Depth的Panel遮住）。

（3）尽量不要让Panel之间共享同一个Depth，这样会导致性能消耗增加。

（4）制作Panel和UI控件时，记得考虑一下它所属的panel和它自身的Depth是否能让它显示在正确的层次关系上。

### 2.7.2 小心相机的深度

我们在场景中的每一个Camera也有一个渲染深度，如图2.36所示。



▲ 图2.36

在前文中我们学习到NGUI创建时，都会创建一个它独有的相机。这个相机其实就是Unity中普通的Camera，然后为其附加了一个UICamera的组件。需要注意的是，所有的Camera也都有一个Depth，这个Depth会影响到UI中的Depth，特别是场景有多个Camera来渲染不同层次的UI时，这个影响会比较大。具体我们得遵循以下这些规律。

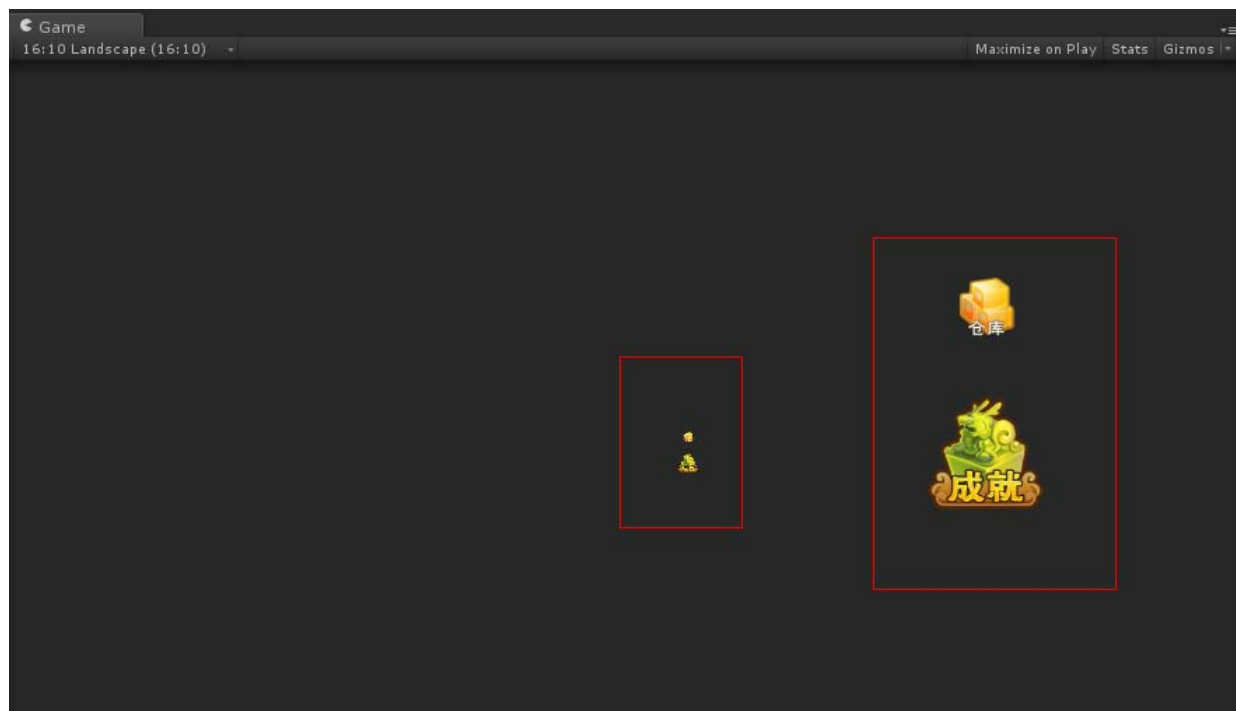
(1) 相机的Depth永远是最高级的，也就是高Depth相机所看到的画面，永远在低Depth相机所看到的画面之上。

(2) 如果需要相机有视觉穿透效果（只渲染所看到的東西，其他地方透掉显示其他相机所看到的画面），需要将相机的ClearFlags设置为DepthOnly。

(3) 并不是只有负责渲染NGUI的相机的Depth会有影响，所有的相机（比如默认存在的渲染场景的MainCamera）的Depth都受此规律影响。例如，如果将照射UI的摄像机的深度设为0，然后将照射场景的相机深度设为1，那么，将看到场景把所有的UI遮住。

(4) 创建UI时，UIRoot下生成的相机默认Depth是比场景中的相机深度高的，不过当场景内有多个摄像机时，一定要管理好每个摄像机的ClearFlags和Depth。

(5) 当场景内有多个摄像机时，一定要检查摄像机的CullingMask不要渲染重复的Layer，否则可能导致显示双重画面。如图2.37所示，UI画面被两个相机同时看到，显示了两份（因为两个相机所在的位置不一样，所以看到的大小会不一样）。



▲ 图2.37

## 第3章 核心组件

### 3.1 什么是UI控件

UI控件这个词是我们制作UI过程中经常碰到的一个词，那么到底什么是UI控件呢？具体来说，UI控件是指一个界面中可以操控的元件，如按钮、输入框等。但是在开发游戏的过程中，一般来说UI控件可以指界面中所有的UI元件，包括精灵、贴图、标签、输入框、下拉列表、按钮等，大家不需要对这个概念深究，在使用NGUI时，凡是带有Widget参数（后文要讲）的组件都是控件。

### 3.2 制作精灵（UISprite）

#### 3.2.1 怎样判断是否应该使用精灵

在一套UI中，精灵是一种非常常见的元件。当我们制作UI时，如果需要显示一张图片，需要先判断这个图片是否应该制作到图集里去，然后用精灵的方式去使用它，一般来说，可以遵循以下规律。

（1）首先说明一点，精灵是一个很基础的UI元件，经常不会独立使用，很多其他控件都会用到精灵，比如一个进度条，需要用到一个表示空槽的精灵，和一个上面走进度的条子精灵。所以，精灵有的时候并不是独立使用的。

(2) 对于一些展示型的图片，不会变化，只是起一个展示作用，例如，一个界面上的一个小花纹装饰，如果它不大，它一般都是以精灵的方式去制作。

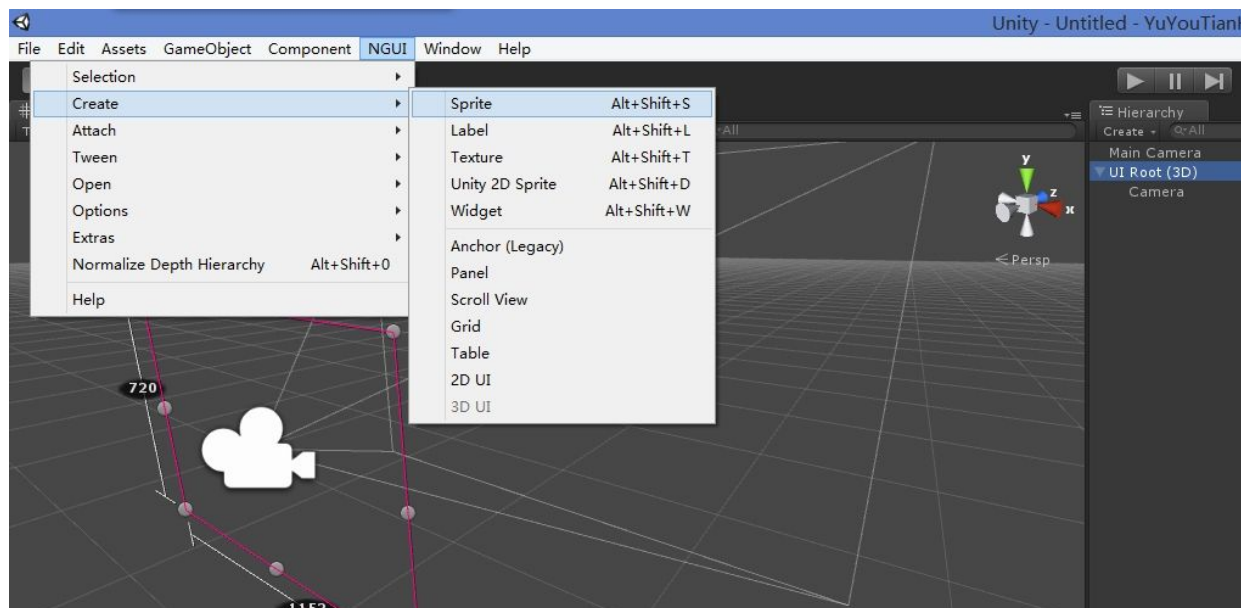
(3) 如果要显示一个图片，它形状不规则，长宽不是2的N次方，那么一定要使用精灵。因为Unity对非2的N次方的图片处理要慢很多。

(4) 如果这个UI元件经常性地出现，那么最好使用精灵，因为，这样它就可以和图集一起被载入内存，并不用新增一个DrawCall去渲染它。

### 3.2.2 创建精灵

#### 1. 第一种创建方式

首先，我们创建一个 3DUI（2DUI 也行，这个没有任何影响），然后选中 UIRoot，单击Unity顶部菜单中的NGUI菜单，选择Creat，然后选择Sprite，如图3.1所示。这样就能在UIRoot下面自动创建一个带Sprite组件的物体，这就算创建成功了。



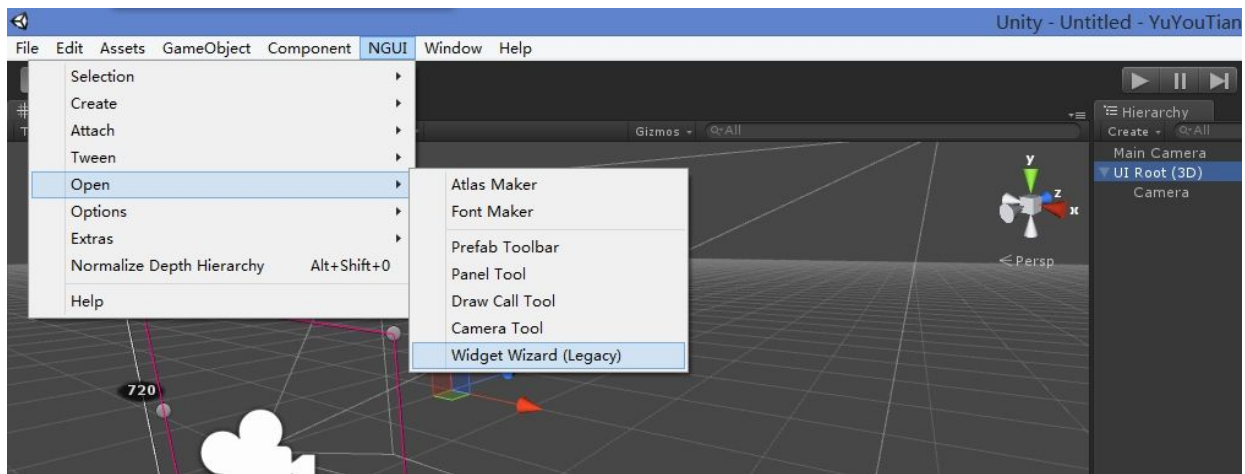
▲ 图3.1



特别说明一下，NGUI创建物体时会在你选中的那个UI物体（可视为一个节点）下进行创建，如果你没有选中任何的UI节点，它会默认在UIRoot下创建。创建出的UI控件的本地坐标都为0（相当于Reset了一下，和父节点的位置保持一致），所以，使用3DUI的时候要注意，不要在UIRoot所附带的Camera下面创建UI元件，否则会导致UI和相机在一个位置，导致相机看不到。

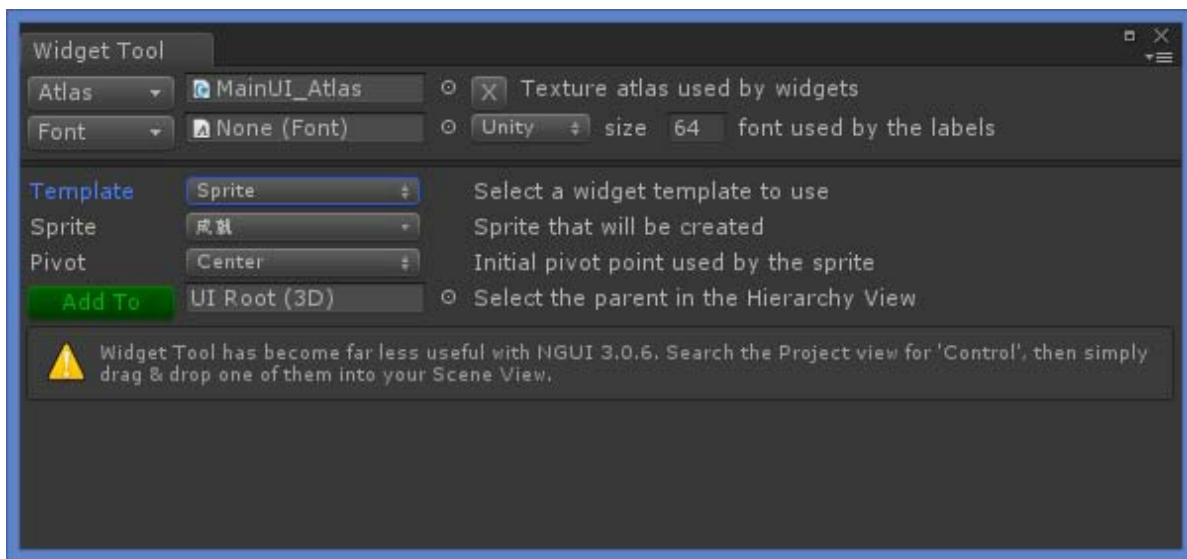
## 2. 第二种创建方式

使用旧版本的创建方式，在 Unity 顶部菜单中选择 NGUI 菜单，选择 Open，选择WidgetWizard(Legacy)，如图3.2所示。



▲ 图3.2

打开后，会弹出如图 3.3 所示的界面，其中选择想要创建的精灵所在的 Atlas，然后在Template中选择Sprite，在Sprite栏单击会弹出你所设置的图集中的所有精灵，从中选择你要创建的精灵，Pivot是精灵的锚点（中心点的位置，默认在图片中心点）。AddTo是选择你要在哪一个UI节点下进行创建（可以通过拖动的方式将UI节点物体拖到这里来），这个AddTo的默认值是你在打开这个菜单之前所选中的UI节点物体。然后单击AddTo按钮，即可完成创建。

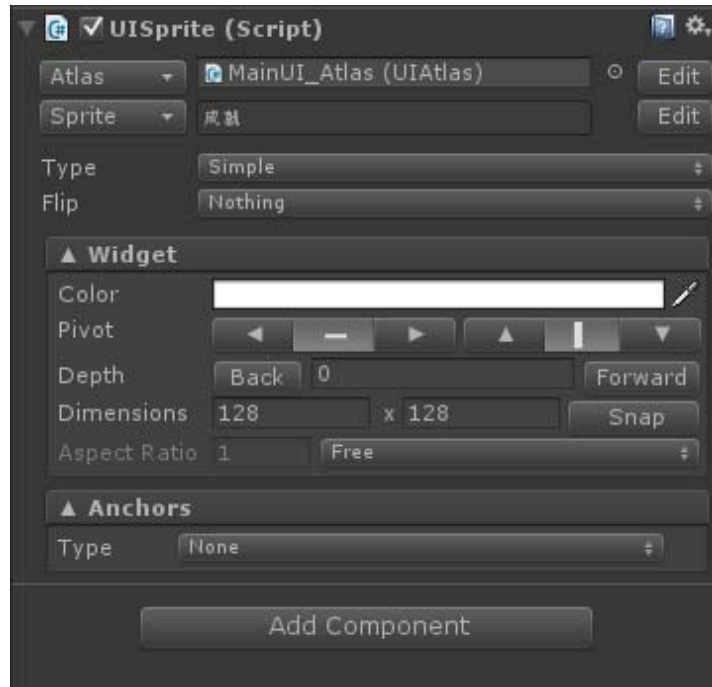


▲ 图3.3

### 3. 第三种创建方式（不推荐）

这种方式是不用NGUI的菜单来创建，通过Unity的空物体，然后为其附加相应组件来自制UI控件。

首先在Unity顶部菜单中选择GameObject，然后选择CreatEmpty，这样就在场景中创建了一个空物体，然后将它的名字改为 Sprite（这个物体的名字可自由定义），再将这个空物体拖动到UIRoot下，使它成为 UIRoot下的一个子物体，将这个空物体的transform组件Reset一下，这样这个物体就和UIRoot根节点保持一样的位置了。然后将这个空物体的Layer改为和UIRoot的Layer一样，否则UI摄像机将无法渲染它。在这个空物体的Inspector面板中，单击Add Component按钮，选择NGUI，选择UI，再选择NGUI Sprite，就为这个空物体附上了Sprite组件，如图3.4所示。



▲ 图3.4

我们在这个**Sprite**组件中单击第一行的**Atlas**按钮，选择要创建的精灵所在的图集，然后单击第二行的**Sprite**按钮，会弹出这个图集所有的精灵预览界面，从中选择所要的精灵。到此为止，精灵就自制完成了。

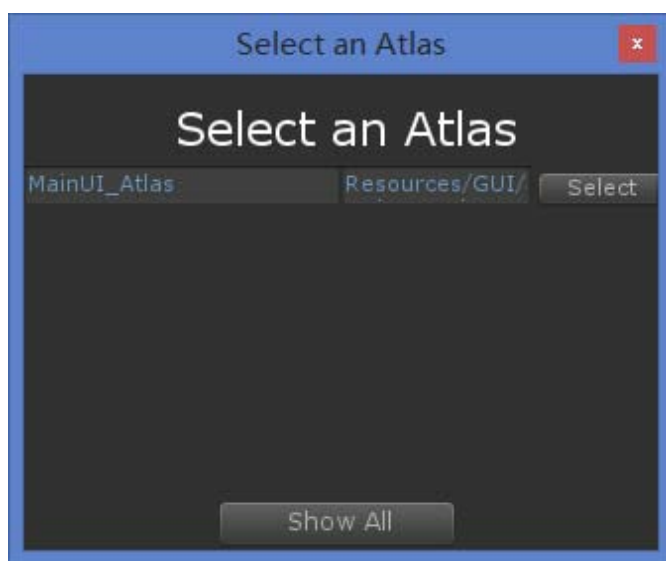
关于创建精灵的方式我们就介绍这3种，它的核心就是要为一个物体附加一个**Sprite**的组件。在制作过程中需要注意创建 **Sprite** 的节点，也就是它应该在什么物体下创建，这个涉及UI结构的设计，我们在后面会详细讲解。

### 3.2.3 Sprite组件的设置

我们参照图3.4所示的组件面板进行讲解。在**Sprite**组件面板中，可以设置如下的一些参数。

(1) **Atlas**。单击**Atlas**按钮将会弹出图集选择界面，如图3.5所示，可以选择要使用哪一个图集（如果弹出的图集选择界面没有我们

要的图集，记得单击该面板中的ShowAll按钮）。



▲ 图3.5

(2) **Sprite**。单击 **Sprite** 按钮，将会弹出该图集所拥有的精灵的预览界面，我们只需要在其中找到需要的精灵，然后双击，就完成了设置。

(3) **Type**和**Flip**。在这里**Type**有5个选项：**Simple**（普通类型）、**Sliced**（切片类型）、**Tiled**（平铺类型）、**Filled**（填满类型）、**Advanced**（高级类型）。**Flip**选项是翻转选项，相应的**Type**下有不同的设置。

#### ●Simple

这种类型下，图片会正常显示出来，图片是什么样它就是什么样显示。当我们将一个精灵的尺寸拉大时，它会以原图拉伸（可能会导致原图发生形变）的方式来完成，如图3.6所示，我们将精灵的大小通过拉动四周的蓝色锚点拉大，精灵就被拉伸了。

在这种类型下，**Flip**有几个选项，分别是：**Nothing**（不翻转）、**Horizontally**（水平翻转）、**Vertically**（竖直翻转）、**Both**（既水平又竖直翻转）。

这里的翻转和Photoshop中的图片翻转是一个意思，如图3.7所示。

### ●Sliced

切片风格，这种类型知识量比较大，和九宫格的制作联系比较紧密，所以在后文讲解九宫格的制作时，会详细讲解。



图3.6



▲ 图3.7

### ●Tiled

平铺类型，选择了之后，精灵尺寸会保持原来的尺寸不变，然后将精灵的尺寸拉大时，精灵会以平铺的方式来填充，并不会以拉伸的方式来填充。如图3.8所示，我们将精灵的大小通过拉动四周的蓝色锚点拉大，精灵变成了平铺模式。



▲图3.8

### ●Filled

填满模式，这种模式可以设置图片填充一块区域的方式，例如，技能CD时技能图标前面有一层灰色的图片蒙住，这个灰色的图片要顺时针旋转消失，一直到转完为止灰色的蒙层彻底消失、图标恢复正常表示CD完成。

在Filled 模式下，会多出Fill Dir、FillAmount、Invert Fill3 个设置项。其中FillDir是指选择填充的方式，默认为 360°填充。Fill Amount 可以设置填充的比例，默认为 1 全部填充。InvertFill是设置填充的方向，不勾选是正方向，勾选是反方向。

如图3.9所示，图中1部分FillDir设置为Horizontally水平填充，FillAmount=0.5，InvertFill不勾选默认为从左至右为正方向，图片相当于从左往右水平填充了0.5，也就是50%





▲ 图3.9

图中2部分为FillDir设置为Vertically竖直填充、FillAmount=0.5, InvertFill不勾选默认从下至上为正方向, 图片相当于从下往上竖直填充了0.5, 也就是50%。

图中3部分FillDir设置为Radial90填充（90°旋转填充）、FillAmount=0.5, InvertFill不勾选默认从右下角到左上角90°旋转为正方向, 图片相当于从右下角到左上角旋转填充了0.5, 也就是50%。

图中4部分FillDir设置为Radial180填充（180°旋转填充）、FillAmount=0.25, InvertFill不勾选默认从左下角到右下角180°旋转为正方向, 图片相当于从左下角到右下角180°旋转填充了0.25, 也就是25%。

图中5部分FillDir设置为Radial360填充（360°旋转填充）、FillAmount=0.65, InvertFill不勾选默认为逆时针360°旋转为正方向, 图



片相当于以中心点为中心逆时针旋转填充了0.65，也就是65%。

#### ●Advanced

高级设置因为比较复杂，将会在后面讲九宫格的时候详细讲解。

#### （4）Widget模块。

Widget模块是NGUI的控件组件都具有的一个模块，如图3.10所示。该模块的参数设置如下。

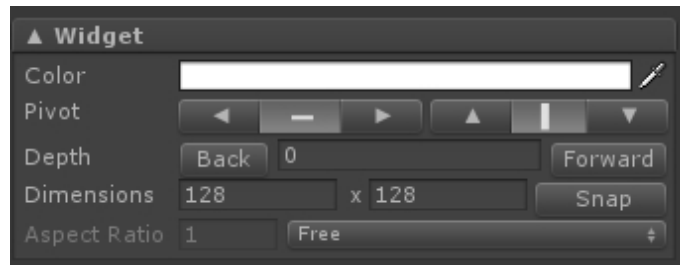


图3.10

#### ●Color

通过这里可以整体改变控件的颜色和透明度，改变颜色的规则为：原色值乘以这里设置的色值（Unity中，会把RGB色值从0~255转变为0~1的一个浮点数）。

我们单击这个白色区域会弹出调色板，如图 3.11 所示。可以随意地在这里设置控件的颜色值和透明度。

值得注意的是，如果通过这个控件改变了透明度，那么这个物体的子物体的控件透明度也会跟着被改变。

#### ●Pivot

锚点设置，默认为中心点。通过这一排按钮可以设置出左上、顶中、右上、中左、中心、中右、左下、底中、右下一共9个点。

这个锚点设置，改变的是图片的中心点位置，这个UI控件和其他UI控件之间的相对位置就是以这个点作为标准的。



▲ 图3.11

### ●Depth

深度设置，前文已经详细讲过。可以通过单击Back和Forward来减1和加1，也可以直接输入一个深度数字来完成设置。

### ●Dimensions

尺寸，这里指的是控件的像素尺寸。单击Snap可以将图片的像素尺寸直接设置为原大小（这个图片被改成图集之前的图片大小）。

### ●AspectRatio

宽高比，AspectRatio 后面的数字为当前该控件的宽高尺寸比例。后面有一个模式选择按钮，默认为Free，可为图片随意设置高和宽。这里除了Free以外，还有两个模式：以宽为基础、以高为基础。如果选择以宽为基础，那么图片的高度设置不论怎么设置都无效，都会以宽度和当前的宽高比计算得出。同理，如果选择了以高为基础，那么图片

的宽度就无法被设置，它的宽度都会以高度和当前的宽高比计算得出。

(5) Anchors模块。

这个模块是控件位置适配的锚点设置，在后面讲解屏幕自适应时会详细说明。

## 3.3 制作标签 (Label)

### 3.3.1 怎样判断是否应当使用标签

当游戏中出现需要程序输出文字的地方，就要使用标签。例如，物品的名字千变万化，无法提前预知有哪些名字，就需要程序人员做一个Label来动态显示物品的名字。

### 3.3.2 创建标签

标签的创建方式也多种多样，具体和Sprite的创建差不多，这里就不浪费篇幅讲解了。一般来说，可以直接在Unity顶部选则NGUI菜单、选择Creat、选择Label，即可创建一个Label。

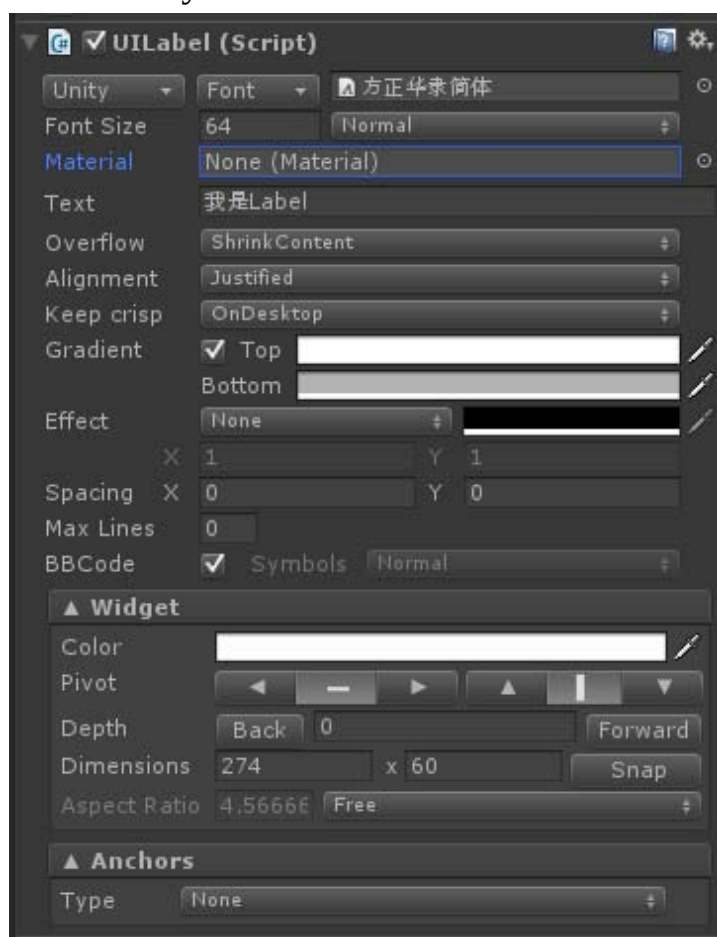
### 3.3.3 Label的文字设置

Label组件的Inspector面板如图3.12所示，我们来了解一下它的各项设置。

#### 1. 设置字体

如果新创建的Label的组件面板为一片灰色（不可设置）的话，说明还没有设置字体。单击图3.12中Unity按钮，会弹出两个选项：NGUI

和Unity，如果希望使用NGUI的静态字体，则选择NGUI；如果希望使用动态字体，则选择Unity。



▲ 图3.12

然后单击Font，如果之前选择的NGUI，那么这里会弹出所有的静态字体以供选择。如果之前选择的是Unity，那么这里将会弹出所有的动态字体文件，以供选择。

如果在其中没有找到你制作或者导入的字体，记得单击ShowAll。

## 2. 设置字号

可以在FontSize中设置我们希望文字的字号大小。但是文字真正显示出来的大小还要受overflow设置的影响（下面讲解）。

在动态字体模式下（选择的Unity中导入的字体文件），FontSize后面有一个字体模式的设定，默认为普通状态。其中可以设置字体

为：**Bold**（加粗）、**Italic**（斜体）、**BoldAndItalic**（加粗并斜体）。

### 3. 设定字体内容（Text）

我们可以在Text选项中，输入需要它显示的文字，支持回车换行。

### 4. Overflow充满设置

要小心这个设置，因为字体虽然设置了字号，但是每一个Label其实依然是一个控件，它也有尺寸。如果字体的字号大小导致字体超出了这个控件的尺寸，这里的Overflow 设置就会对字体进行处理。

#### ●ShrinkContent

收缩内容。默认为这个选项，意思为不管字体的字号设为多大，只要它超出了这个控件的尺寸，就将文字缩小到尺寸范围内。

#### ●ClampContent

选择这个设置意味着如果文字的字号大小导致文字超出了控件的尺寸，就将不显示文字。

#### ●ResizeFreely

选择这个设置意味着不管控件多大尺寸，只要文字字号设定了，文字会保持这个字号应有的大小，然后控件会自动依照文字的大小调整宽高尺寸。

#### ●ResizeHeight

选择这个和ResizeFreely类似，只不过这个选项只会去自动调整控件尺寸的高度，并不会让控件尺寸的宽度变大。

### 5. Alignment

这里是设置对齐方式，一共有：**Auto**（自动，一般会设为居中）、**Left**（左对齐）、**Right**（右对齐）、**Center**（居中）、**justified**（调整，会自动变换）。

这里的对齐和居中的参照标准是控件的尺寸，也就是说左对齐，其实是对齐到这个Label控件的最左边。如果选择了 **justified**，那么文字

会在控件尺寸缩小到一定范围时，自动增大文字的间距来使文字刚好充满它。

#### 6. Keepcrisp

字面翻译为保持脆性，默认为OnDesktop。如果选择Always，则当字体缩小时会变模糊，一般情况我们必须要去设置它。虽然能带来一些性能优化，但是非常渺小。

#### 7. Gradient

梯度，可以理解为字体的渐变，默认为勾选状态。如果勾选，则字体从上到下会有一个渐变，在后面Top和Bottom两个色板中可以设置上部分和下部分渐变的颜色。如果不选择这个选项，那么字体将不再有渐变色，Top和Bottom将不可用，此时字体的颜色将完全地以该控件的颜色为准。

#### 8. Effect

字体的效果设置，一共有3个选择：None（无效果）、Shadow（阴影效果）、Outline（描边效果）。如果选择了阴影或者描边效果，可以在后面的色板中设置阴影或者描边的颜色，并可以在下面的X和Y中设置阴影和描边的XY厚度（约等于像素单位）。

#### 9. Spacing

字体间距，可以设置X（字间距）和Y（行间距）的间距。

#### 10. Maxlines

最大行数。后面的BBcode选项可以不用管。

#### 11. Widget和Anchors

Widget在Sprite精灵的讲解中已经讲过，这里是完全一样的。Anchors在后文讲适配的时候一起讲。

### 3.4 制作UI纹理（UITexture）

### 3.4.1 什么情况下使用UITexture

UITexture 的功能是在屏幕上显示一张图片，在这一点上它和Sprite有着相似的功能，但是UITexture会消耗单独的DrawCall去渲染，并会单独加载进内存，所以，会增大性能的开销。当我们判断是否应该使用UITexture时，可以遵循以下规律。

(1) 当图片过大，不适合成图集时，可以使用 UITexture，此时要尽量保证图片的宽高是2的N次方（宽高不必相等，不过在iOS平台下必须宽高相等才能支持压缩）。

(2) 当图片尺寸为2的N次方，但出现频率不高时，可以使用UITexture。例如，游戏的Logo，一般出现它都是在游戏开始的时候偶尔出现一下，此时可以使用UITexture。

(3) 修改更换特别频繁的图片，为了减少每次更新维护的麻烦，可以考虑使用UITexture。

(4) 如果图片很小，尽量将图片放入图集通过精灵的方式使用。

### 3.4.2 创建纹理

UITexture的创建方式和Sprite、Label等一样，通过Unity顶部的NGUI菜单，选择Creat进行创建。其他创建方式就不做介绍了。

### 3.4.3 纹理的设置

UITexture的Inspector组件设置界面如图3.13所示，我们来了解一下UITexture组件的各项设置。

#### 1. Texture

纹理设置，将要显示的贴图文件拖到此处即可完成设置。

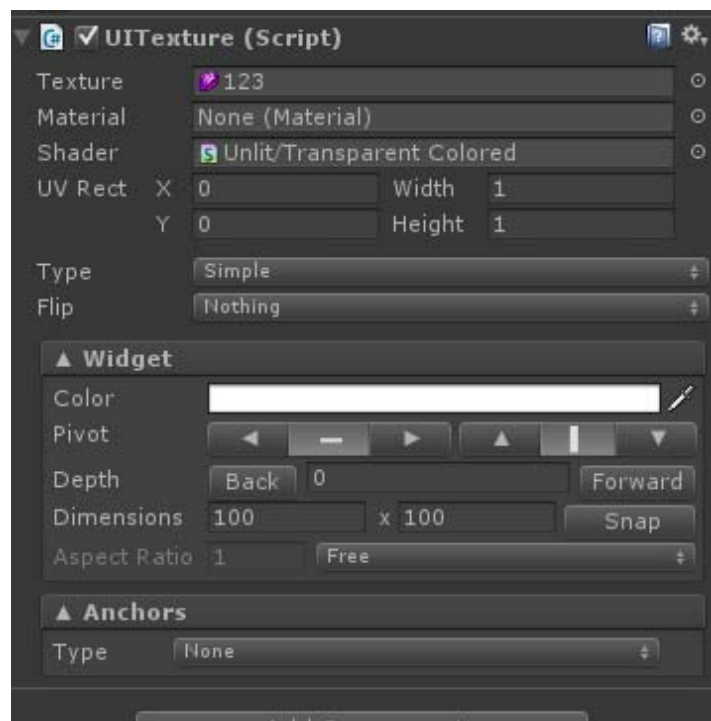
#### 2. Material



材质设置，一般不用去设置它，如果有特殊材质需求可以拖到这里来。

### 3. Shader

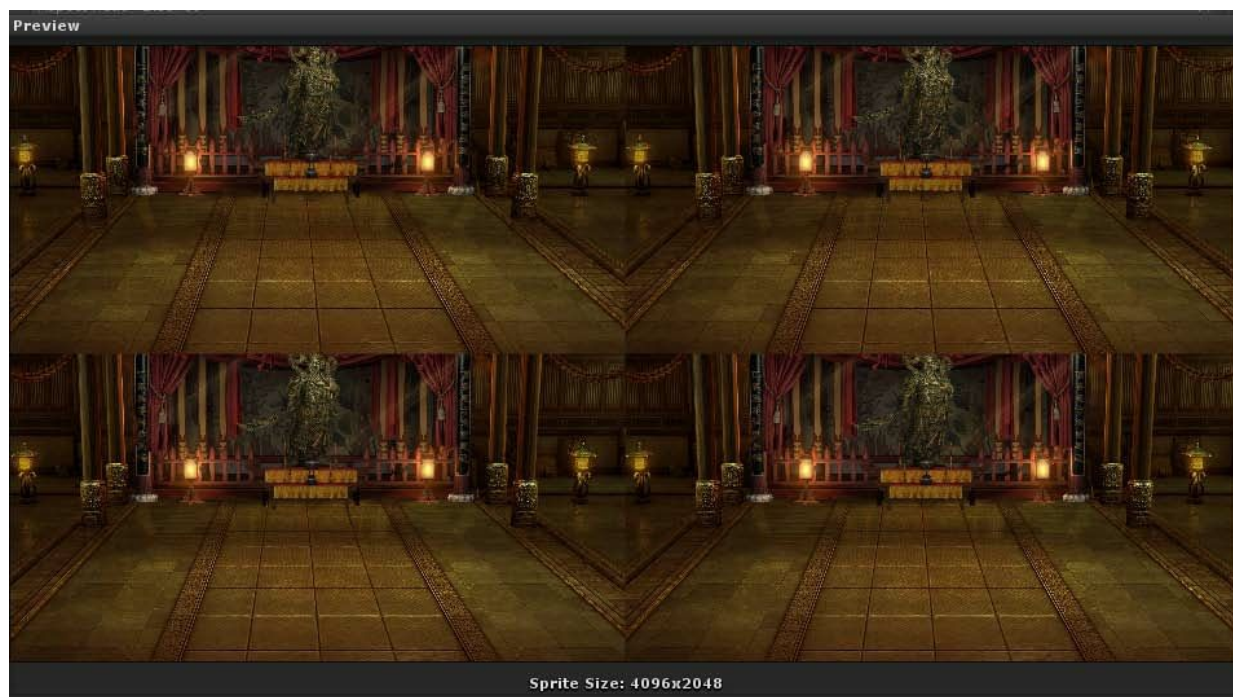
着色器设置，默认为带透明的颜色贴图着色方式，如果有特殊的着色需求，可以将其Shader拖到这里，不过，一些特殊的Shader将大幅增加性能开销，要谨慎使用Shader。



▲ 图3.13

### 4. UVRect

UV矩形的设置，如果在width和height中各填为2，那么将会是4张纹理拼在一起。如图3.14所示，一般游戏开发中，这个UVRect都不需要进行设置。



▲ 图3.14

## 5. Type/Flip

这个和Sprite一样，请参看前文对Sprite的讲解。

## 6. Widget/Anchors

略。

# 3.5 制作按钮 (Button)

## 3.5.1 怎样判断应该使用按钮

按钮是我们制作UI过程中最核心、最重要的控件之一，在一个游戏中，按钮是用户操作最依赖的控件之一，几乎无处不在。按钮的核心作用就是：能接收我们的单击事件，然后触发一个响应事件。

当我们在判断是否应该使用按钮时，可以先了解按钮的核心作用。

(1) 按钮能接收单击并触发响应事件。  
(2) 按钮被单击时能同时触发多个响应事件。  
(3) 按钮可以有普通、悬停、单击、禁用等多个状态的不同表现。

(4) 不用去区别一个按钮是普通Button还是ImageButton，在新版NGUI中这俩已经合并，只有一种Button了。

(5) 广泛的说，按钮的核心在于接收事件，任何可以接收用户操作事件的，都可以称之为按钮。

图3.15展示了一些按钮的案例。



▲ 图3.15

### 3.5.2 创建按钮

在旧版本的NGUI中，按钮分为了两种按钮：普通Button（在一个背景底板上面写有按钮的文字）和ImageButton（按钮是一张纯图片，并且不同状态可以变为不同的图片）。旧版本中的按钮创建方式也是由WidgetWizard来创建。但是，将普通的按钮和ImageButton分开并不是很科学，因为在游戏中，这两种按钮的应用太广泛了，每次都需要去区分创建普通Button和ImageButton是一件很麻烦的事。所以，在新版本的NGUI中，普通Button和ImageButton合并了。

创建按钮，可以用以下任一种简单的方式。

(1) 创建一个Sprite，这个Sprite将会是按钮的外形。

(2) 选中创建的这个Sprite，然后在Unity顶部菜单中选择NGUI、选择 Attach、选择 Collider。

(3) 选中创建的这个 Sprite，然后在 Unity 顶部菜单中选择 NGUI、选择 Attach、选择ButtonScript。

我们可以用Label、Texture等其他控件来代替Sprite去制作一个按钮，方法一模一样，并不一定非要以Sprite作为基础进行创建。只是用Sprite制作按钮在游戏开发中更为普遍，所以以它作为例子。

如果需要在创建好的这个按钮上写上文字，例如，“确定”、“取消”等，只需要选中这个按钮，然后在它下面创建一个Label，并写上“确定”即可，注意，Label的深度要高于这个按钮的深度，这样就完成了一个“确定”的按钮。

小提示，创建出来的 Sprite 记得单击 Snap，让它回归到原尺寸大小，然后再去等比例调整它的大小，这样可以尽量减少图片的形变。

### 3.5.3 核心组件BoxCollider

首先，按钮要接收单击事件，必须要有一个控件形态，它可以是Sprite、Label、Texture等。至于这个组件的使用，我们前文讲过了，就不浪费篇幅多讲了。我们直接讲解按钮的另外两个核心组件。

#### 1. 核心组件：BoxCollider

BoxCollider是一个物理组件，准确地说是一个物理碰撞盒，所有的需要接收外部输入事件的（如单击、拖动等）UI，都需要拥有一个BoxCollider，这个BoxCollider代表的是响应事件的范围。如果没有BoxCollider，那么这个控件无论如何都无法接收到外部事件，这是NGUI的一个底层原则。

既然BoxCollider代表的是接收事件的响应范围，那么，如果我们将一个按钮的BoxCollider大小设为全屏幕，则单击屏幕上任何一个地方，都相当于单击了这个按钮。

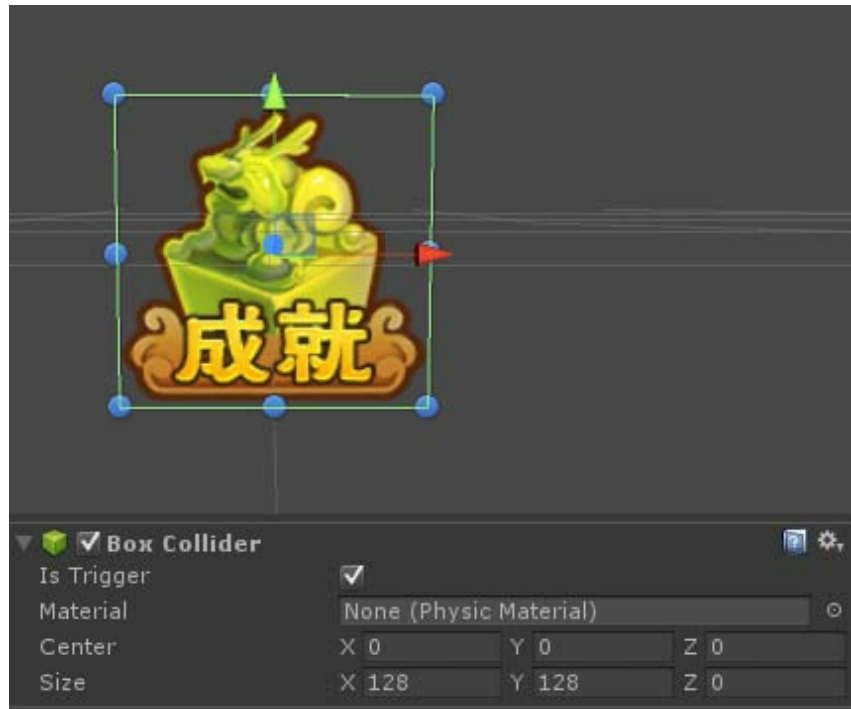
我们可以通过上一节讲的Attach方法自动生成一个BoxCollider，也可以通过Inspector面板中单击 AddCompent 手动附加一个 BoxCollider。区别在于：通过 NGUI 菜单 Attach 的BoxCollider的大小，会自动刚好包围控件的范围，而手动创建的BoxCollider，大小需要手动去调整。

BoxCollider的组件设置如图3.16所示，图3.16中按钮的边框变成了一个绿框，这个绿框就是BoxCollider的包围框，只要在Inspector面板中展开BoxCollider的组件，就能看到这个绿框，如果Inspector面板中BoxCollider组件的设置菜单被收起来了，则不会出现这个绿框。

在BoxCollider中，我们可以看到它的设置很简单，先来熟悉一下它的设置。

#### （1）Is Trigger。

是否打开触发器，这个设置对于NGUI没什么用，它打开的作用是可以通过物理碰撞触发事件（如相撞爆炸等）。



▲ 图3.16

### (2) Material。

材质设定，这里设定的是物理材质，对NGUI也没有什么用，它的作用是为这个碰撞盒包围的物体设定一个物理的表面，例如，一块地面是草地、还是木板、还是金属。

### (3) Center。

中心位置的偏移。BoxCollider 都有一个中心点，这里的 Center 就是设置它的中心点相对于控件的中心点的偏移，是一个相对量，所以需要注意一点：BoxCollider 的这个 Center会受到控件本身的Pivot中心点设置的影响。如图3.17所示的BoxCollider的Center偏移量为（0，0，0），但是，因为控件的Pivot将中心点设置为了右上角的点，所以形成了图3.17所示的情况。





▲ 图3.17

#### (4) Size。

尺寸设置。这是一个非常重要和常用的设置，经常配合 **Center** 设置一起使用，以此来调整控件响应区域的大小和位置。例如，假设我们要做一款手机游戏，在屏幕中有一个很小的关闭按钮，经常导致玩家点不中它，我们就可以依靠设置 **Size** 来将它的响应区域变大，这样玩家只要单击到关闭按钮周围的区域，都能触发关闭按钮的响应事件。值得注意的是，这里**Size**的Z值在UI中是几乎没有用的，**X**和**Y**的值都是以像素为单位。

需要注意的是，**BoxCollider** 一般需要依赖于一个非空的、实质的物体，例如，如果这个**BoxCollider**物体身上没有控件（**Sprite**、**Label**、**Texture**等），只有一个孤零零的**BoxCollider**，那么在大部分情况下，这个**BoxCollider**是无法接收事件的。

另一个核心组件如3.5.4节所讲。

### [3.5.4 核心组件UIButton](#)

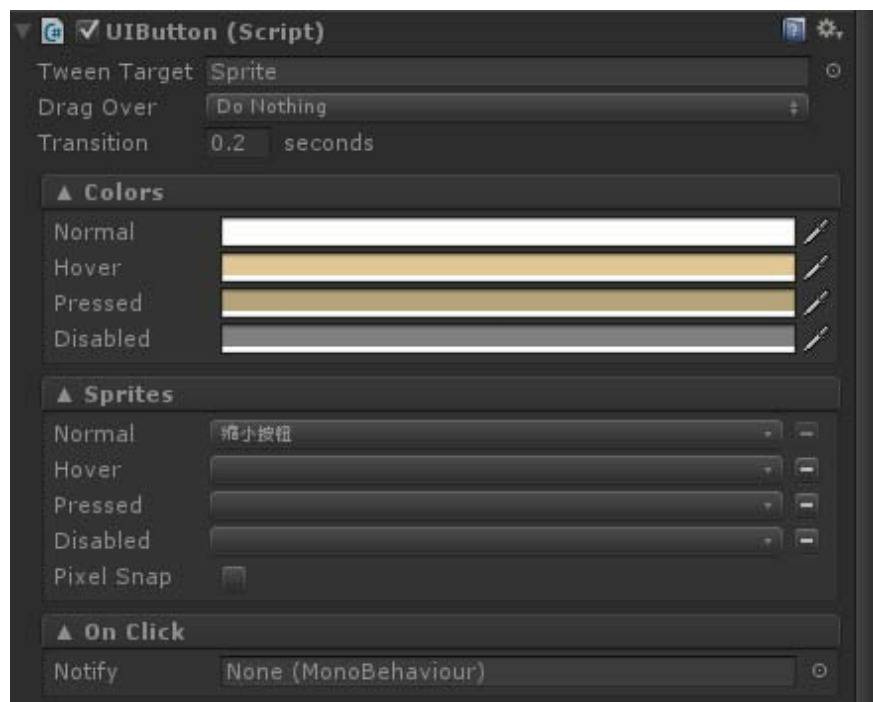


首先我们要明确的是，UIButton 并不是一个按钮的必须组件，也就是说，没有 UIButton 时，我们也能完成一个按钮。只要这个控件上有 BoxCollider，我们就可以在脚本中通过OnClick()、OnHover()等事件监听函数去触发一个响应事件。那么，UIButton的价值到底是什么？什么情况下使用UIButton？UIButton有以下一些重要的用途。

(1) 可以设置不同状态下的颜色，比如普通状态、单击状态、鼠标光标悬停状态、禁用状态，可以有不同的颜色来表示。

(2) 可以设置不同状态下的图片，这就是所谓的ImageButton，比如我们希望图片A在鼠标光标悬停在上面时，变成图片B。

(3) 可以很方便地被动态赋予响应事件并分发事件。也就是说，我们可以临时地在 A 物体身上的脚本里动态让B按钮拥有一个或者很多个完全不同的响应事件。以前的旧版本NGUI中，都是用EventListener来实现，新版本中将会更加方便和快捷。



▲ 图3.18

我们来了解一下按钮组件的设置，UIButton按钮组件的Inspector面板设置界面如图3.18所示。

#### （1）TweenTarget。

动画目标，默认为按钮自己。按钮在光标悬停时变色、被单击时变换图片等，都是动画。绝大部分情况下，这个设置都需要设为自己（默认就是按钮自己）。

#### （2）DragOver。

拖动结束事件，默认为Do Nothing。这里有两个选项：Do Nothing和Press。之所以有这个选项，是因为按钮在被拖动时，有一个事件的交叉：如果我们拖动一个按钮，那么不仅仅拖动了它，同时也按下了它。这个设置的目的是，定义它被拖动结束后，是否还执行按下事件。

#### （3）Transition。

过渡时间。这里是动画过渡的时间，例如，我们设定按钮在鼠标光标滑过时，要变黑，这个设置就是设置它在光标滑过时由正常到变黑的时间，

#### （4）Colors模块。

这里是改变颜色，可以设置在不同状态下按钮的颜色和透明度。一共提供了4种状态可设置：普通状态、按钮被鼠标光标滑过时的状态、按钮被按下的状态、按钮不可单击时（BoxCollider被禁用）的状态。设置颜色的方式前文已讲过，就不再赘述。

#### （5）Sprites模块。

这里是精灵设置模块，也就是整合进来的ImageButton模块。值得注意的是，如果制作按钮时不是使用一个精灵控件作为基础制作的，那么将不会有这个模块。例如，我们为一个Texture附加BoxCollider和UIButton来制作按钮，此时UIButton组件就不会有这个模块。

在这里可以设置按钮在不同的状态下显示什么样的图片。一共支持4种状态：普通状态、按钮被鼠标光标滑过时的状态、按钮被按下的状态、按钮不可单击时（**BoxCollider** 被禁用）的状态。

当我们通过**Sprite**制作一个按钮时，我们创建的**Sprite**会默认出现在**Normal**设置中，然后任何状态下，按钮图片都是这个精灵不会再变化。

如果我们需要按钮在不同状态下进行变化，我们可以分别对每个状态需要显示的精灵进行设置，设置的方法为单击状态名称的按钮，会弹出图集中所有精灵的预览（图集是**Normal**状态显示的精灵所属的图集）。

如果我们要取消一个状态下显示精灵的设置，可以单击该状态后面的小减号。但是，普通状态下的精灵无法被取消，当其他状态没有设置精灵时，将会默认显示普通状态的精灵。

**PixelSnap** 是指保持原像素尺寸，当我们设置了不同状态下显示不同的**Sprite** 时，只有勾选了这个框，才会让显示的不同的**Sprite**都以原像素尺寸显示。如果不勾选，则所有状态下的**Sprite**都会统一以这个按钮控件（制作按钮时所用的**Sprite**）的尺寸大小进行显示。

#### （6）OnClick。

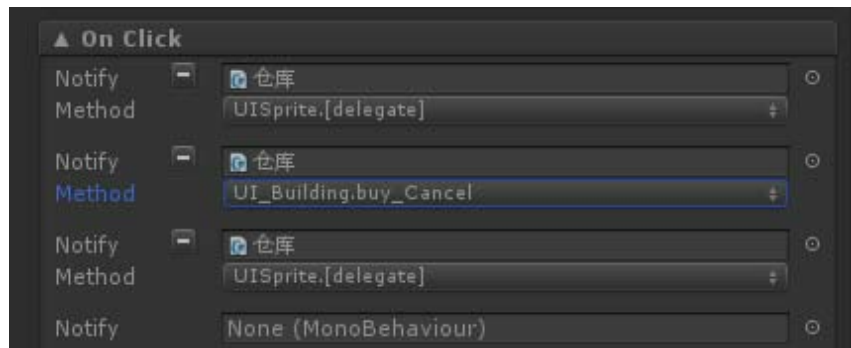
这里是设置按钮响应单击事件的地方。我们设置按钮响应事件时可以通过以下操作。

将该按钮要响应的事件函数的脚本所在的物体拖入到 **Notify** 中，然后该物体的名称就会出现在**Notify**的设置框中，会自动出现该物体下所有脚本中的**Public**函数，然后选择要按钮响应的那个函数即可。

也就是说，在这里可以让按钮响应任何一个物体身上的、任何一个脚本的、任何一个**Public**函数。

在这里当你设置了第一个响应事件之后，会自动弹出第二个响应事件的设定，也就是说，通过这里的设置，可以让按钮被单击之后响

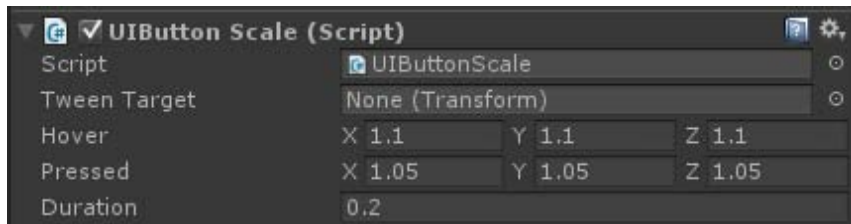
应任何多个事件，如图3.19所示。



▲ 图3.19

### 3.5.5 制作按钮的放缩动画

如果我们希望单击按钮时，按钮会有一个放缩动画（如突然变大并蹦一下），我们可以单击AddComponent、选择NGUI、选择Interaction，在里面找到ButtonScale脚本，附加到按钮物体上，如图3.20所示。



▲ 图3.20

ButtonScale的核心作用就是控制按钮的放缩动画，我们来了解一下它的设置。

#### (1) Script。

这个是它所调用的脚本，我们不用管，它会自动定位好。

#### (2) TweenTarget。

这个是它所控制的动画作用的目标物体，我们不用管。在运行时，它会自动设定为当前所属的按钮物体。

#### (3) Hover。

当鼠标光标划过时，按钮控件的大小变化。

(4) **Pressed**。

当按钮按下时，按钮控件的大小变化。

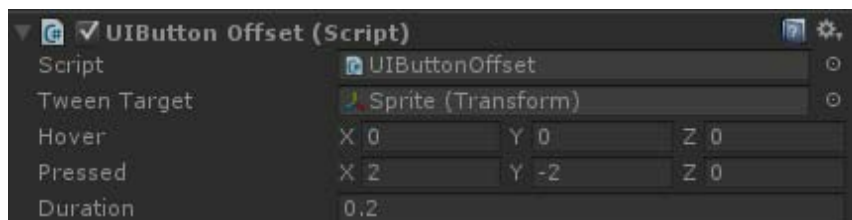
(5) **Duration**。

完成缩放动画的时间，以秒为单位，默认为0.2秒完成。

需要注意的是，这个组件一般只用于按钮，如果非按钮要做动画，请参看后文讲解的Tween动画制作。

### 3.5.6 制作按钮的偏移动画

如果希望按钮有一个偏移动画，例如，单击按钮时按钮会向右下方蹦一下，我们可以为按钮制作一个偏移动画。方法和制作缩放动画类似，在AddCompent菜单中，选择NGUI、选择Interaction，然后找到ButtonOffset，附加到按钮上。



▲ 图3.21

图3.21就是ButtonOffset组件的设置界面，我们来了解一下它的设置。

(1) **Script**、**TweenTarget**和**ButtonScale**组件一模一样，都不用管这俩设置。

(2) **Hover**。

按钮在鼠标光标滑过时的位置偏移。这里偏移的是相对坐标。

(3) **Pressed**。

按钮在按下时的位置偏移，这里也是相对坐标。

(4) **Duration**。

持续时间。

### 3.5.7 制作按钮的旋转动画

如果我希望按钮在被单击时能旋转一下，就可以为按钮制作一个旋转动画，方法和制作缩放偏移动画一模一样，旋转动画的脚本是 **ButtonRotation**。具体设置和其他动画类似，这里就不多赘述了。

### 3.5.8 添加按钮单击音效

在游戏开发中，一般情况下我们都需要为按钮增加单击音效，为按钮添加音效的办法很简单，可以在 **AddCompent** 中，选择 **NGUI**、选择 **Interaction**，然后将 **PlaySound** 组件附加到按钮上。



▲ 图3.22

图3.22是 **PlaySound** 组件的设置界面，我们来了解一下它的设置。

(1) **AudioClip**。

音效的源文件，将音效文件拖到此处即可。

(2) **Trigger**。

触发模式，就是在什么情况下触发音效，默认为 **OnClick**。这里给了以下几种模式：**OnClick**（单击触发）、**OnMouseOver**（鼠标光标移上来）、**OnMouseOut**（鼠标光标移开触发）、**OnPress**（按下触发）、**OnRelease**（释放触发）、**Custom**（自定义触发，即脚本中控制触发）。

(3) **Volume**。

音量大小，为0到1之间的一个浮点数。

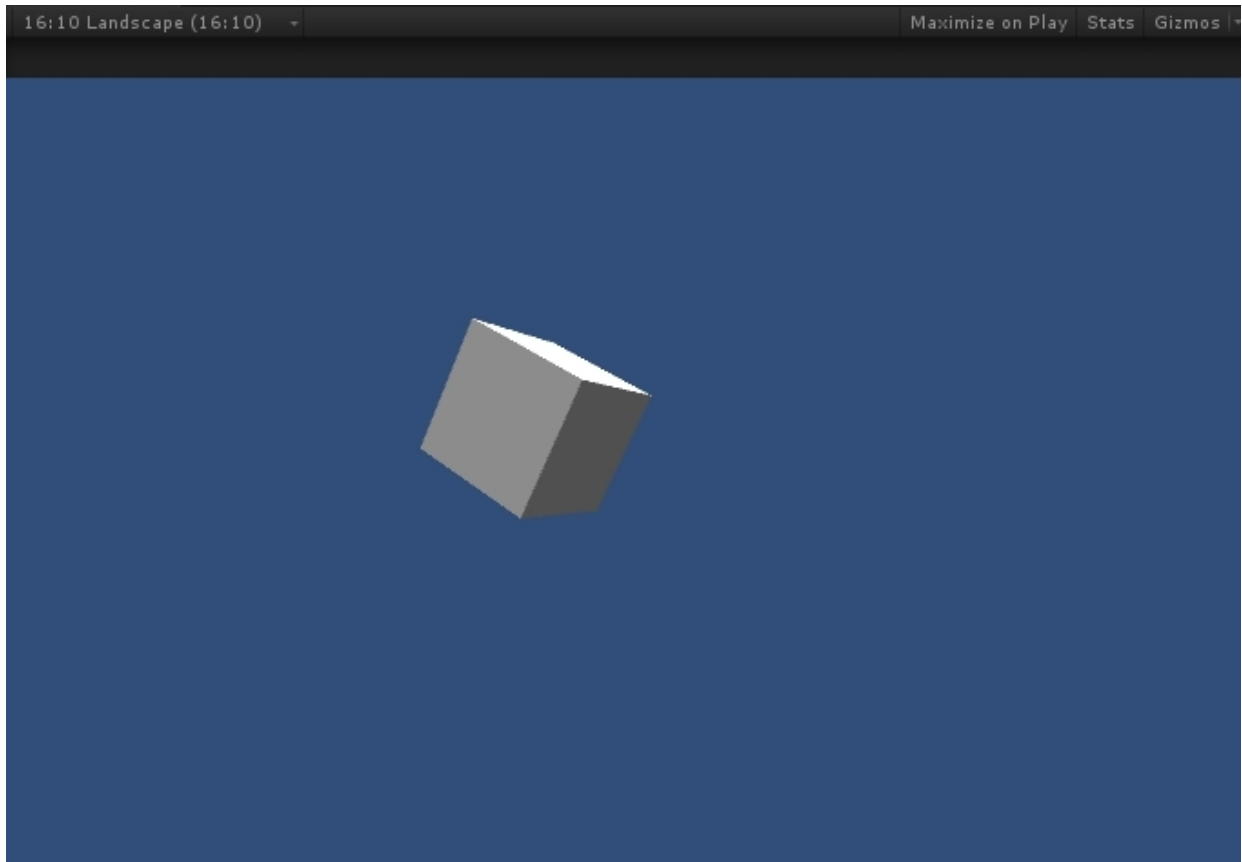
(4) Pitch。

音调，也为0到1之间的一个浮点数。

### 3.5.9 任何事物都可以变成按钮，不仅仅是UI

利用按钮的原理，可以将任何事物变成一个按钮。下面举个例子：我们把一个三维模型变成按钮，让它可以接收事件。

首先，我们单击Unity顶部菜单的GameObject，选择Creat、选择Cube创建一个立方体模型（也可以从外部导入模型文件，这里为了讲解方便，直接采用简单的立方体）。为了让它看得更清楚，可以加一个灯光，创建好之后如图3.23所示。



▲ 图3.23

然后我们要进行如下的关键步骤。



### 1. 让摄像机能够监听事件

根据前文所讲，只有带有UICamera组件的摄像机照射到的物体，才能接收事件响应。我们可以看到这个Cube是由场景中MainCamera渲染出来的，我们先为MainCamera附加一个UICamera组件。

选中MainCamera，在Inspector面板中单击AddCompent按钮，选择NGUI、选择UI，然后选择那个NGUI Event System，这个组件就是UICamera 组件。这样我们就为主摄像机附加了NGUI的事件监听。

### 2. 为Cube附加按钮的关键组件BoxCollider

选择Cube，首先为其附加一个BoxCollider。可以通过AddCompent按钮选择Physics，然后BoxCollider（也可以通过Unity顶部NGUI菜单去Attach一个BoxCollider）。

默认情况下这个 BoxCollider 的尺寸为 0，我们需要为它调整尺寸，值得注意的是，这个Cube因为不是UIRoot的子物体，所以，这里的尺寸单位是：米。我们将尺寸设为（1,1,1），然后就能看到Cube被一个绿框所包围。设置尺寸的时候要注意，这个Cube并不是一个UI图片，所以一定要设置Z轴的尺寸。

### 3. 为Cube附加UIButton

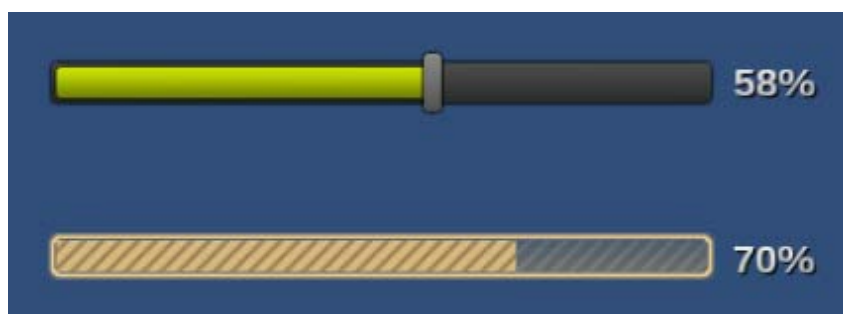
附加方法可以通过Unity顶部NGUI菜单去Attach，也可以通过AddCompent。制作好之后，我们会发现这个Cube之下并没有ImageButton才拥有的Sprites模块，前文我们有讲过，只有以Sprite作为基础制作出来的按钮才会有这个Sprites模块。

到此为止，我们就已经将Cube变成了一个按钮，鼠标光标移动上去，会发现它颜色改变为Hover状态下的颜色了，说明它已经接收事件了。

## 3.6 制作进度条（UISlider）

### 3.6.1 怎样判断是否应当使用进度条

之前我们介绍了Sprite、Label、Texture、Button的用法，之前的4种都是比较常用的基础控件。而进度条，是一种更为高级的控件，它是多个控件的结合体。用进度条的主要目的是为了用一根管子的充满程度来直观地表示某种数值的百分比，进度条分为可拖动和不可拖动两种。图3.24所示为两种进度条，上方的为可以拖动的进度条，下方的是不能拖动的进度条。



▲图3.24

可拖动进度条和不可拖动进度条的原理几乎是一模一样，唯一的区别是可拖动进度条上多了一个拖动块和BoxCollider来接收事件，而不可拖动的进度条只能显示一个数字的百分比，无法由玩家去操控。

我们在判断是否应该使用进度条时，有以下的规律可以遵循。

(1) 如果某一种值，它有最大值，我们需要表达它当前的值的占比，这个时候用进度条会非常直观。此时应当用不可拖动的进度条。例如：角色的生命值、法力值、角色升级经验等。

(2) 如果某一种值，它有最大最小值，我们希望玩家去自由拖动设置，如音量调节、亮度调节等，我们就可以使用可拖动的进度条。

需要强调的是，可拖动进度条和不可拖动进度条，他们的原理是一模一样的，本质区别只是可拖动进度条有一个BoxCollider。它们都有三大核心要素构成：底槽Sprite、进度条Sprite、滑动块。

### 3.6.2 创建进度条

进度条因为是多种基础控件的一个集合体，所以创建方式有多种。下面就介绍两种创建方式：自己拼装和使用预设。

#### 1. 第一种方法：自己拼装出一个进度条

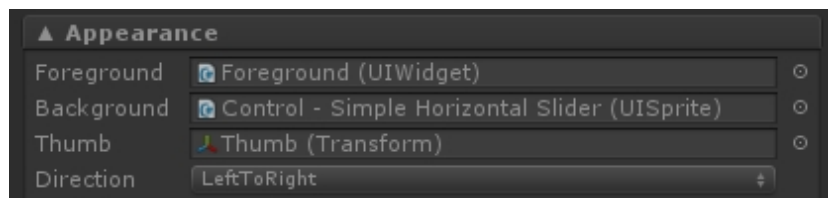
(1) 创建一个底槽Sprite。

(2) 为底槽 Sprite 附加一个 UISlider 组件。附加方法为 AddCompent → NGUI → Interaction → NGUI Slider。

(3) 在底槽Sprite下创建一个进度条Sprite作为子物体，调整好尺寸用以和底槽相吻合。

(4) 在底槽 Sprite下创建一个滑动块Sprite作为子物体（如果不需要拖动可以不创建这个物体，省去这一步），然后在底槽Sprite上 Attach一个BoxCollider。

(5) 将底槽Sprite拖动到自身UISlider组件上的Background选项中，将进度条Sprite拖动到底槽的UISlider组件上的Foreground选项中，将滑动块Sprite拖动到底槽的UISlider组件上的Thumb选项中，这样三大要素就齐备了，如图3.25所示。



▲ 图3.25

(6) 如果希望显示当前进度的百分比数字，则在滑动块下创建一个Label（如果不希望数字的位置跟着滑动块走，也可以在别的地方创建Label），然后将该Label物体拖动到底槽的UISlider组件的 OnValue Change 模块下的 Notify 中，然后在出现的 Method 选项中选择 UILabel.SetCurrent Percent方法。

一个进度条到此就创建完成了。

## 2. 第二种方法：使用PrefabToolBar直接创建

在新版本的NGUI中，它自身制作了一些常用的UI控件的预设，当我们需要使用时，直接拖动预设到场景中，就可以直接完成创建。

在Unity顶部菜单中选择NGUI菜单，选择Open、选择PrefabToolBar，然后会弹出如图3.26所示的界面，这里面就是NGUI已经制作好的一些预设。



▲ 图3.26

拖动其中想要的预设到UIRoot下（或者其他的UI节点下），就可以完成创建了。创建完成之后记得去修改它的各个Sprite为你所要用的Sprite。

因为NGUI创建的预设是固定的，具有一些缺点，比如UI结构不一定符合我们实际项目的需求，比如某些组件我们并不需要。所以，当你足够熟悉理解和运用NGUI的情况下，在实际的游戏项目开发中，建议自己制作进度条。

### 3.6.3 核心组件UISlider设置

对于一个进度条控件来说，不管是不是可以拖动的进度条，它的核心组件是UISlider脚本，它的组件设置界面如图3.27所示：

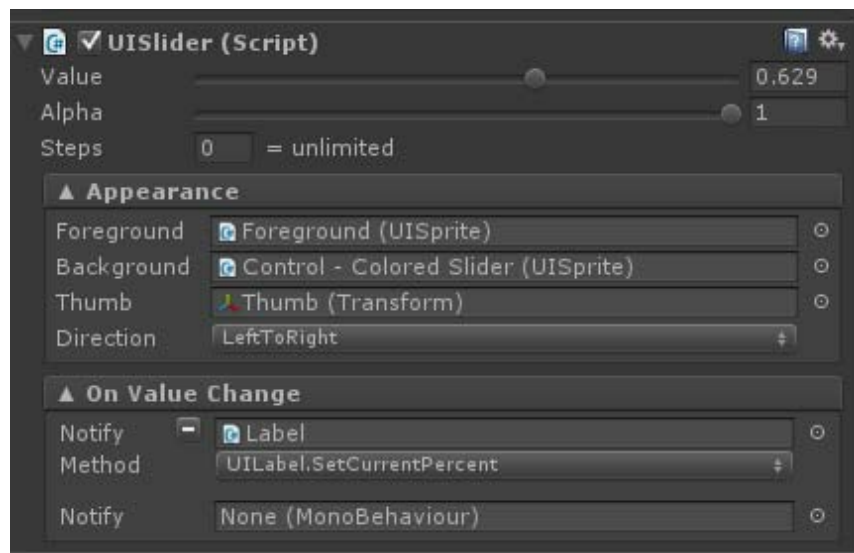
我们来了解一下它的工作原理。

### 1. Value

进度值，这是为了显示当前数值在“总槽”里的百分比，一般我们制作游戏时，都会在代码中调用这个参数，让它由游戏中相应的数据动态计算得出来。

### 2. Alpha

透明度，默认为 1。这里会控制整个进度条控件的透明度，可以用来做一些淡入淡出的效果。



▲ 图3.27

### 3. Steps

每次变动的步伐大小。默认为0，0就是无限制，也就是Value值可以是任意一个值，如果设置了，那么Value就会“一段一段的”变化。一般游戏开发中，我们都不需要设置这个值。

它的填值效果为“关键点数量的概念”，例如，填入5，则代表完整进度条只有5个点，相当于进度条的值将会只有：0、0.25、0.5、0.75、1一共5个值。

### 4. Appearance模块

这里是设置滑动条的一些组件。

- Foreground

这是进度条上层表示进度的Sprite，将它拖动到这里就算完成了设置。Foreground的长度会随着Value的变化而自动变化。

- Background

这是进度条的底槽 Sprite，将它拖动到这里就算完成了设置。底槽的长度是不会发生变化的。

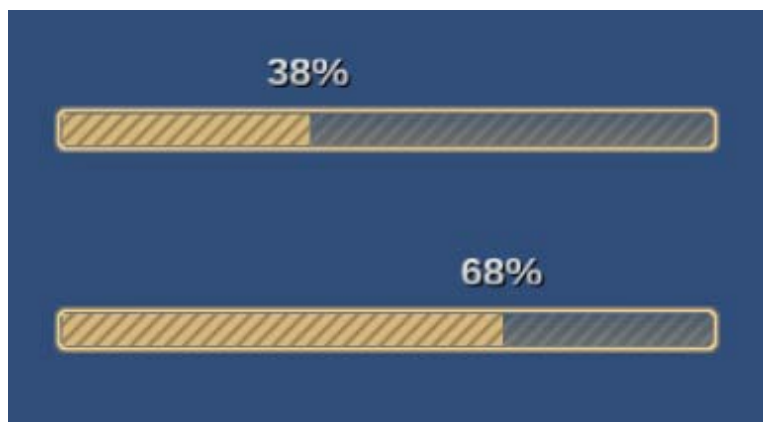
- Thumb

这是拖动块的设置，将任何一个物体拖动到这里来，它就将随着Value的变化而发生位置的变化。

例如，如果希望在进度条的当前进度位置显示一个数字，就可以将这个数字作为 Thumb拖到这里。然后数字的位置就会跟着进度条移动，永远处于进度条的当前进度位置。如图3.28所示，Thumb数字的位置会随着Value变化，永远显示在当前进度的位置。

- Direction

进度条的正方向，默认为从左至右。里面提供了4种选择：从左至右、从右至左、从上到下、从下到上。



▲ 图3.28

## 5. On Value Change

这是进度变量发生变化时的一个回调函数，当Value值发生变化时，就会执行这里的函数。这里的设定方法，和设置Button的回调方法



是一模一样的。

值得注意的是，如果我们希望在值发生变化时，自动改变一个百分比数字（Label）的显示，NGUI为我们提供了一个简单的方法：将要显示该进度条百分比的Label物体拖入到Notify中，然后在Method栏中选择UILabel.SetCurrentPercent方法，这样，当进度条的Value值改变时，它就能自动地改变这个Label文本的显示。

### 3.6.4 进度条的BoxCollider说明

- BoxCollider只有附加到底槽上才有用。
- 如果没有BoxCollider，进度条无论如何都无法进行拖动设置。
- BoxCollider将会接收进度条上任何一个位置的消息来直接设置进度。例如，不去拖滑动块，直接在90%的位置点一下，那么进度会直接变为90%。
- BoxCollider和拖动块Thumb没有必然联系，如果没有BoxCollider，那么即使有拖动块，也无法通过拖动和单击等来设置进度。
- 只要有BoxCollider，即使没有拖动块，我们也能直接拖动和单击来设置进度位置。

## 3.7 制作输入框（Input）

### 3.7.1 怎样判断是否应当使用输入框

输入框，就是用户可以自由输入文本的地方，输入框因为其功能的单一性非常好判断是否应当使用输入框。当我们需要判断是否需要



使用输入框时，可以遵循一条原则：凡是需要用户自主输入文本的地方，几乎都必须使用输入框。

输入框的常见用法：输入登录账号和密码、输入角色名称、输入聊天内容等。

### 3.7.2 创建输入框

我们来学习创建一个如图3.29所示的输入框。



图3.29

首先，我们要了解输入框控件的 3 个核心控件：**BoxCollider** 组件允许 UI 能输入事件、**UIInput**组件允许玩家能输入自己的文字、一个**UILabel**来显示输入的文字。

#### 1. 第一种创建方式：自己拼装

(1) 创建一个**Sprite**作为输入框的底板（就如图3.29中的底框）。

(2) 为这个输入框的底板附上**UIInput**组件，附加方法为

**AddCompent** → **NGUI** → **UI** → **Input Field**。

(3) 为这个输入框附加一个**BoxCollider**，附加方法为先选中底板**Sprite**，然后选择**Unity**顶部**NGUI**菜单并选择**Attach** → **BoxCollider**。或者**AddCompent** → **Physics** → **BoxCollider**（此种方式创建的需要自主调整**Box**的尺寸用以匹配输入框大小）。

(4) 在这个输入框下面创建一个**Label**子物体，创建方法为选中输入框，然后选择**Unity**顶部**NGUI**菜单，选择**Creat** → **Label**即可。

(5) 将这个新创建的**Label**子物体拖入到底框的**Input**组件中的**Label**选项（**Input**组件的第一个选项）中。

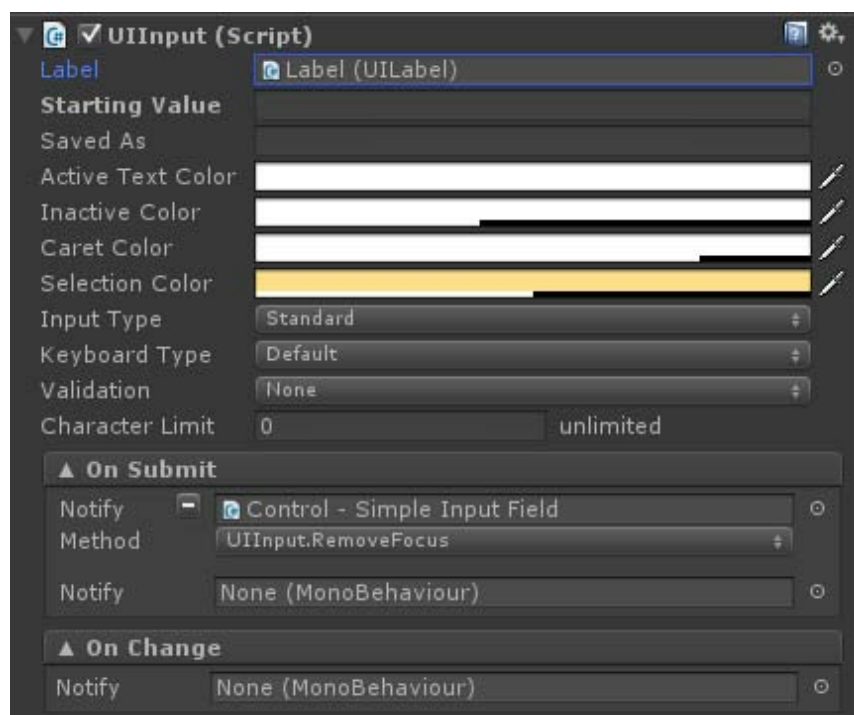
#### 2. 第二种创建方式：**Prefab Tool Bar**中直接拖入

之前我们讲过，在新版本的NGUI中，它制作了一批常用控件的预设。在Unity顶部选择NGUI菜单，选择Open → Prefab Tool Bar，就能打开预设预览菜单，然后在其中将名为Simple Input Field 的预设体拖入到Hierarchy窗口中我们要创建的UI节点下即可。

### 3.7.3 核心组件Input设置

Input组件是输入框的核心组件，我们之前讲Label时说过NGUI显示文字几乎全部依赖Label组件，所以，Input组件它本身是无法显示文字的，它必须和一个Label有一定的关联，让这个Label来帮助显示Input的文本内容。

Input组件的界面如图3.30所示。



▲ 图3.30

我们首先来熟悉一下Input组件的设置内容。

#### 1. Label设置

这个是最核心的设置之一，刚才我们已经讲过，NGUI中所有的文本显示都依赖于Label，Input本身是无法显示文本内容的，所以，它需要借助一个Label来显示玩家输入的文本，这也是我们之前在学习自己拼装输入框控件时为什么要在输入框下面创建一个Label子物体的用意。

我们将一个Label拖入到这里就算完成了设置，以后这个输入框中玩家输入的所有内容都将显示在这个Label上。

如果在这里我们不设置Label，运行之后单击输入框将会报错导，致输入框无法使用。

## 2. Starting Value

默认输入的文字。这里一定要注意区别“默认输入的文本”和“初始显示的文本”两个概念的区别！

默认输入的文本：当运行游戏时，输入框默认输入的文本，这个文本是模拟玩家输入进去的，就好像登录QQ时自动填充了你的QQ号一样，这个文本是一个属于输入的文本、真实有效的输入文本（只不过它是初始自动就输入进去了而已）。

初始显示的文本：这个是输入框在没有接收用户输入时显示的提示文本，例如：“请输入密码”、“请在这里单击进行聊天”等提示性文字，用户单击输入框之后文字就消失了变成了用户输入的文本的显示。这个初始显示的文本只有一个纯粹的提示作用，不属于输入的文本、是一个无效的文本。

这里的 Starting Value 变量设置的是默认输入的文本。初始显示的文本在 Input 相关联的Label中进行设置，这个关联的Label的文本内容就是输入框初始显示的内容。

## 3. Saved As

输入的内容在Player Pref中的哪个字段保存。这个我们一般用不到，而且它会自动保存，基本不用去管它。如果有特殊需求，可以在

这里设置。

#### 4. Active Text Color

活动文本的颜色和透明度设置，也就是用户在输入文字时的文字颜色和透明度。默认为白色，有需求的可以自行设定。

#### 5. Inactive Color

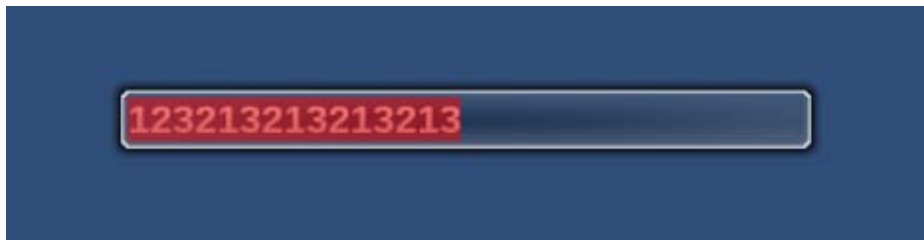
不活动的文字颜色和透明度，这个可以用来设定初始显示的文字的颜色和透明度。

#### 6. Caret Color

这个可以设定插入符的颜色和透明度。

#### 7. Selection Color

选中的颜色和透明度，这个表示的是我们选中输入的文本时，覆盖在文本上的那一层遮罩的颜色，就像在Word办公软件中，我们输入的字可以通过拖动鼠标光标来选中它们一样。如图3.31所示，我们设定了选中之后遮罩为红色。



▲ 图3.31

#### 8. Input Type

输入类型的设定，默认为Standard标准输入。

##### ●Standard

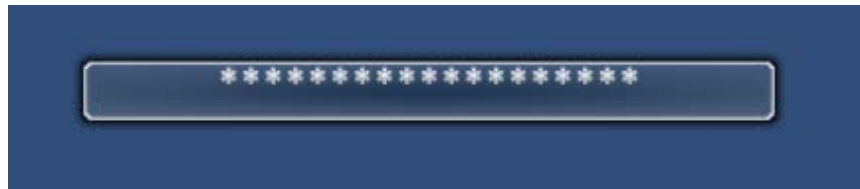
标准输入类型，输入的文字会正常一个挨一个地显示出来。它的对齐方式全部由Input相关联的Label决定。

##### ●AutoCorrect

自动调整模式，类似于Label中的自动调整模式。

##### ●Password

密码模式，在这种模式下，输入的文字会自动地变成\*符号，如图3.32所示。



▲ 图3.32

## 9. Keyboard Type

这是输入文本时，键盘的类型设定。它有以下几种类型：

- Default;
- ASCCapable（任何格式都允许）；
- NumbersAndPunctuation;
- URL;
- NumberPad;
- PhonePad;
- NamePhonePad;
- EmailAddress;
- HiddenInput。

## 10. Validation

验证。默认为none没有验证，可以验证整数、浮点数等，如果输入的字符不属于验证类型将无法输入（整数允许视为浮点型）。

## 11. Character Limit

可输入的最大字符数限制。需要注意的是，一个汉字要占用两个字符。

## 12. On Submit

这里是提交输入内容时的触发事件函数设定。

## 13. On Change

这里是当输入内容改变时的触发事件函数设定。

### 3.7.4 输入框使用的一些注意事项

输入框是一个集成了很多功能的组件，我们在使用输入框控件时，要注意以下一些要点，以免发生不必要的困扰。

(1) 输入框Input本身是无法显示文字的，它必须借助于一个Label来帮它显示输入的文本。

(2) 输入框输入文本的字体，是在输入框关联的Label中设定的，这个关联的Label用的什么字体，则输入框中输入的内容就会是什么字体。

(3) 输入框必须要有一个BoxCollider和一个Sprite底框，否则无法输入。

(4) 输入框中对输入的文本的设定都受关联的Label的影响，因为它本身是借助这个Label来显示文字的。但是，如果发生冲突，比如在Input组件中设定文字颜色为红色，而在关联的Label中设定文字颜色为白色，那么将会以Input中的设置为准。

(5) 如果发生以下情况，都将造成输入框无法显示文字：

- 超出最大字符数限定了；
- 输入的字符不符合要求的验证类型；
- 关联的Label所选用的字体中没有这个文字；
- 关联的Label中设定了文字大小超出范围则不显示；
- 将文字设为全透明了。

(6) 输入的文字可以从Input中的value变量读取，也可以从关联的Label中的text变量读取。

(7) 请将相关联的Label设为输入框的子物体，这样就可以保证输入的文字和底框保持相对位置不变。

## 3.8 制作滚动视图 (ScrollView)

### 3.8.1 怎样判断是否应当使用滚动视图

所谓的滚动视图，是指一个可以滑动的视窗，视窗大小和位置固定不变，视窗内的内容用户可以通过手指滑动或者拖动滚动条来进行滚动浏览。比如说浏览器浏览网页时，我们可以通过拖动右侧的滚动条来滚动浏览网页内容。

滚动视图的目的是为了解决同类内容过多，一个UI版面显示不下的情况。如果同类内容过多，我们一般可以采取设置多个页面，然后通过翻页浏览的方式来浏览，但是很明显，滚动视图会比翻页更方便，因为在移动设备上可以很方便地划屏进行滚动，在PC上可以通过鼠标的滚轮进行滚动。

当我们需要判断是否应该使用滚动视图制作UI时，可以遵循以下规律：

- (1) 有很多同类内容一个版面显示不完，却必须要让用户很方便地进行浏览；
- (2) 它的核心目的是方便浏览。

### 3.8.2 创建滚动视图

滚动视图的创建方式有两种：通过菜单直接创建，或自己拼装。它们的基本原理都一样，都是为了凑齐几个核心组件。

我们来了解一下最简单的创建方式。

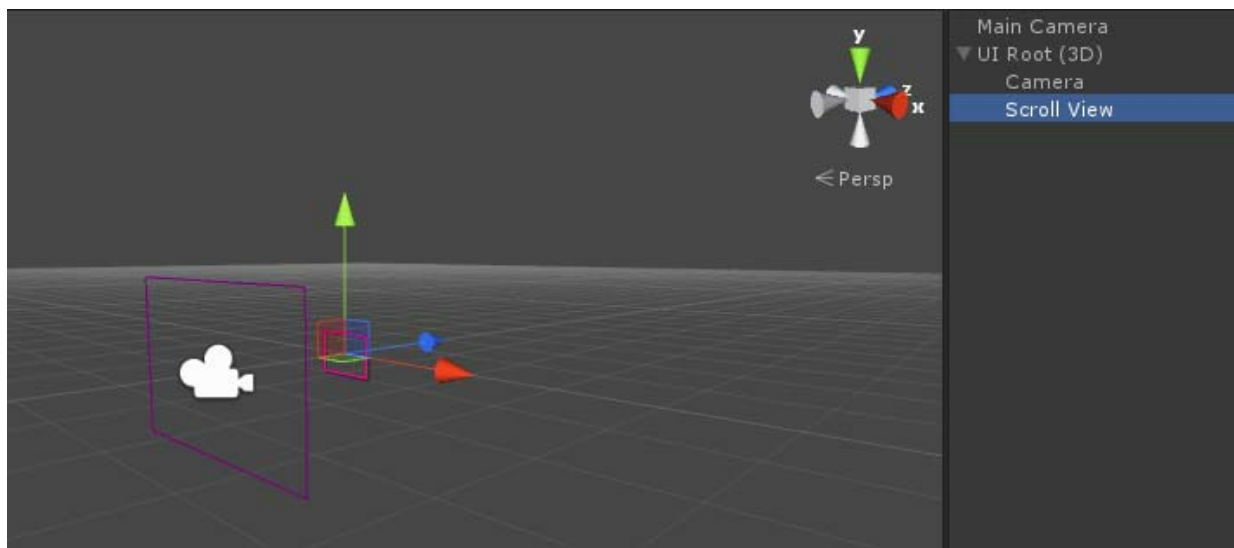
- (1) 选择要创建滚动视图的UI节点，ScrollView将会在这个UI节点物体下作为子物体进行创建。
- (2) 单击Unity顶部 NGUI菜单，选择Creat、选择ScrollView，这样将会自动创建一个ScrollView的子物体在选中的UI节点下。



(3) 到此，我们已经创建完成，我们可以将它的名字改为我们需要的名字方便管理。

目前为止只是创建的一个最基本的滚动视图视窗，现在还无法进行拖动看效果，因为滚动视图是一个相对复杂的UI控件组合，所以，将会分几个章节慢慢讲解它是如何工作起来的。

我们创建完的一个滚动视图的最基础的视窗如图3.33所示，图3.33中选中的那个红框就是滚动视图的视窗。

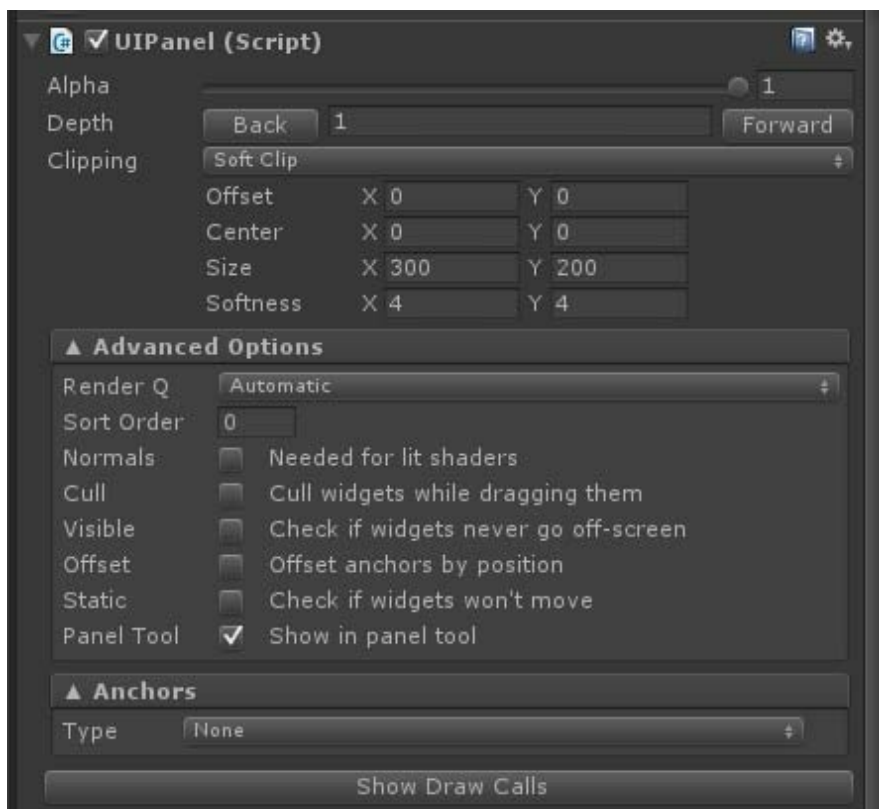


▲ 图3.33

### 3.8.3 滚动视图核心组件UIPanel

我们创建完滚动视图之后，发现它身上自动附带了 3 个组件：UIPanel、UIScrollView、Rigidbody。其中Rigidbody这是NGUI为了优化UI被移动等的性能开销而自动附加的，我们可以不用管它。而另外两个组件UIPanel和UIScrollView则是核心组件，它俩构成滚动视图的基本窗口。

首先我们来了解一下UIPanel在滚动视图中的核心作用，它的组件截图如图3.34所示。



▲ 图3.34

至于UIPanel这个组件的作用，我们在第2章讲解NGUI的几大基础核心组件时已经介绍过了，这里就不多赘述了。我们从图3.34中可以看到，我们创建的ScrollView身上的UIPanel组件有很多地方都自动设定好了，我们来一起了解一下它的工作原理。

(1) 在ScrollView被生成时，为了让它能够在上层显示，所以它自动给Panel设定了一个深度，这个深度大小是当前情况下刚好可以显示在最上层的深度，也就是当前 UIRoot 的 UI树中深度最大的Panel的深度+1。

(2) 我们可以看到它的Clipping被自动设为了SoftClip。之前我们讲过这是UIPanel的面板剪辑，也就是说UIPanel可以让自己的面板上只剪辑一块区域出来进行显示。这样在剪辑区域之外的UI元件就将看不见了。

Clipping一共提供了3种模式。

## 1. None

无剪辑模式，这种模式下，滚动视窗中的物体可以被拖动，但是视窗因为没有剪辑，所以是没有边界的！这将可能导致内容被拖出屏幕外再也拖不回来。就像我们往下拖动浏览网页时会拖到一个所谓的“底”，none模式就是没有这个“底”，你可以将内容全部拖出屏幕以外。

## 2. SoftClip

柔和剪辑模式，我们一般都会使用这种模式来制作ScrollView。

在这种模式下，Panel将会剪辑一块可视区域出来显示，这个被剪辑出来的区域以外的部分将会被剪辑掉而无法显示出来。

在柔和剪辑模式下，我们可以看到以下几个设置项。

### ●Offset

视窗的偏移，以像素为单位，设置这个参数将会导致视窗以Panel的中心点为基准进行偏移。

### ●Center

调整视窗的中心点，效果和Offset一样。

### ●Size

视窗的大小，一般情况下，我们都需要调整视窗的剪辑窗口的大小，以此来匹配背景底板的大小。如果视窗Size比底板大，将会导致视窗内容会滑动出底板的边框，如果比底板小，则会导致视窗内容滑动还没有到底板边缘就已经被剪辑掉消失了。

### ●Softness

剪辑边缘的柔和程度，视窗中，内容被拖动到边缘部分时，会有一个渐隐的效果。如果这个Softness的值设为0，则内容被拖动到边缘时，会像被刀切掉一样被剪辑掉。

如图3.35中红框所示部分则为Softness的X为50的剪辑边缘，我们可以看到X方向的左右两边剪辑的边缘变得更柔和了。



▲ 图3.35

### 3. Constrain but don't Clip

这种模式下是指视窗会尽量地包含所有内容但是不剪辑它们，效果大约等同于有边界但是边界为全屏，无法完全将内容拖到屏幕外面去，只要在屏幕范围内，都能看到内容，内容并不会被剪辑掉。

图3.36则是一颗巨大的桃树图片放在了一个ScrollView中，被UIPanel剪辑了的景象。通过这个图我们可以进一步加深了解UIPanel在滚动视图中的作用。

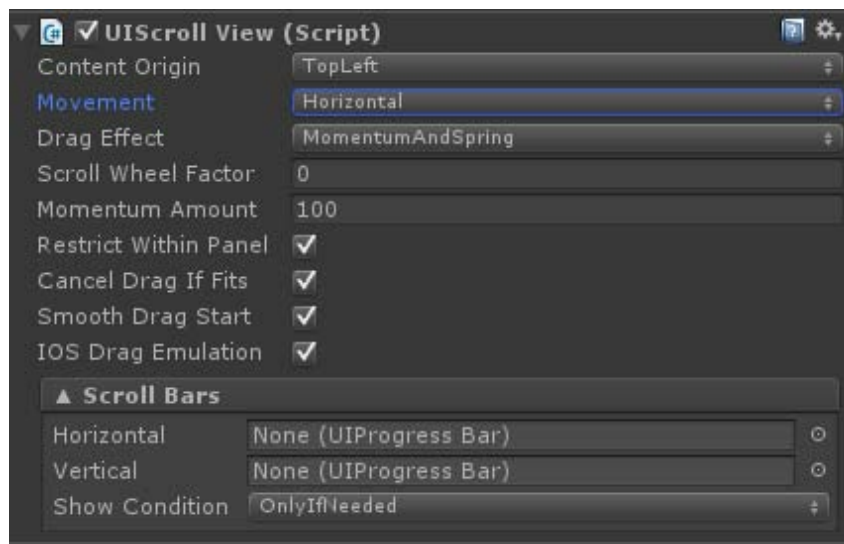


▲ 图3.36

### 3.8.4 滚动视图核心组件UIScrollView

刚才我们了解了UIPanel是滚动视图中控制视窗大小的核心组件，而UIScrollView则是滚动视图中控制滚动功能的核心组件。没有UIScrollView组件的视窗是无法进行滚动的。

我们先来了解一下UIScrollView的组件，组件界面如图3.37所示。



▲ 图3.37

### 1. ContentOrigin

这是滚动视图所包含的内容的起点，默认设置为左上角。

### 2. Movement

滚动视图的滚动方向，也就是内容的移动方向。一共提供了4种方式。

- Horizontal

水平方向拖动，也就是左右。

- Vertical

竖直方向拖动，也就是上下。

- Unrestricted

自由拖动

- Custom

自定义方向，会弹出新的X和 Y设置框，可以通过对 X、Y的设置来定义一个特殊的方向。

### 3. Drag Effect

拖动效果。里面提供了3种效果。

- None

无效果，拖到哪算哪。手指停下或者离开屏幕的一瞬间视图内容也停下了。

- Momentum

带动能的拖动，也就是有一个惯性，当我们手指松开时，视窗内容还会继续朝之前的方向滑动一会儿，中途如果遇到边界才会立即停下，否则会等惯性没了才会自动停下。这种效果主要目的是让用户用最少的操作来滑动视图内容。

- MomentumAndSpring

动能和弹性兼备的一种方式，这种方式和上一种动能方式很相似，区别在于，在这种方式下，当滑动内容拖到了视窗边界时，它可

以被继续拖动“越界”，在松开手指时立即像弹簧一样弹回并回到正常视窗范围内。

#### 4. Scroll Wheel Factor

滚轮因素的设定，这个值越大鼠标滚轮的滚动就会越灵敏。

#### 5. Momentum Amount

动能因素的设定，这个值越大，滑动时的惯性就越大（前提是Drag Effect有动能效果）。

#### 6. Restrict Within Panel

选中后拖动将会被限制在Panel内，这个是默认勾选的。如果不勾选这个选项，则将会导致内容被拖到视窗的剪辑部分以外再也无法回来。

#### 7. Cancel Drag If Fits

当它刚好适合视窗内时，则自动退出拖动。

#### 8. Smooth Drag Start

在开始的时候拖动平滑，勾选上后，我们拖动时内容会有一个速度从0变到我们拖动的那个速度的一种平滑自然的效果。如果不勾选，在我们开始滑动时，内容会瞬间与手指的速度一样，就像黏在手指上了一样，体验比较差劲。

#### 9. IOS Drag Emulation

模拟iOS系统的拖动效果。这也是一种为了增强拖动体验的选项。

#### 10. ScrollBars

为滚动视窗指定拖动条，拖动条我们将会在下文介绍。这个设置是个非必选项，可以设置也可以不设置，不设置的话滚动视窗一样能正常工作，因为在移动设备上，我们可以用手指滑动，在PC上我们可以通过按住鼠标键不放来拖动，而拖动条，可以理解为我们拖动的进度条，就像我们浏览网页时最右边的那根竖直的进度条一样。

拖动条一共可以指定两个，一个水平的、一个竖直的。



我们可以在Show Condition中设定拖动条出现的规则，有以下3种规则可选。

- Always

不管什么情况，滑动条总是出现。

- OnlyIfNeed

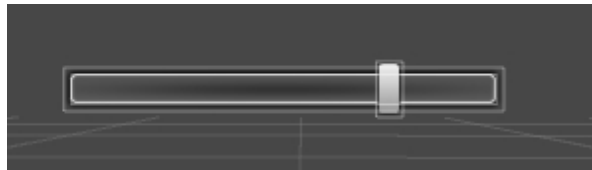
只有当需要的时候出现。那么什么时候是需要的时候呢？当我们的内容在滚动视窗内显示不完的时候，就会出现，如果我们在视窗内不需要拖动就可以看到全部内容，则不需要出现。

- WhenDragging

当拖动时出现。只要拖动内容，它就一定会出现。

### 3.8.5 创建一个拖动条

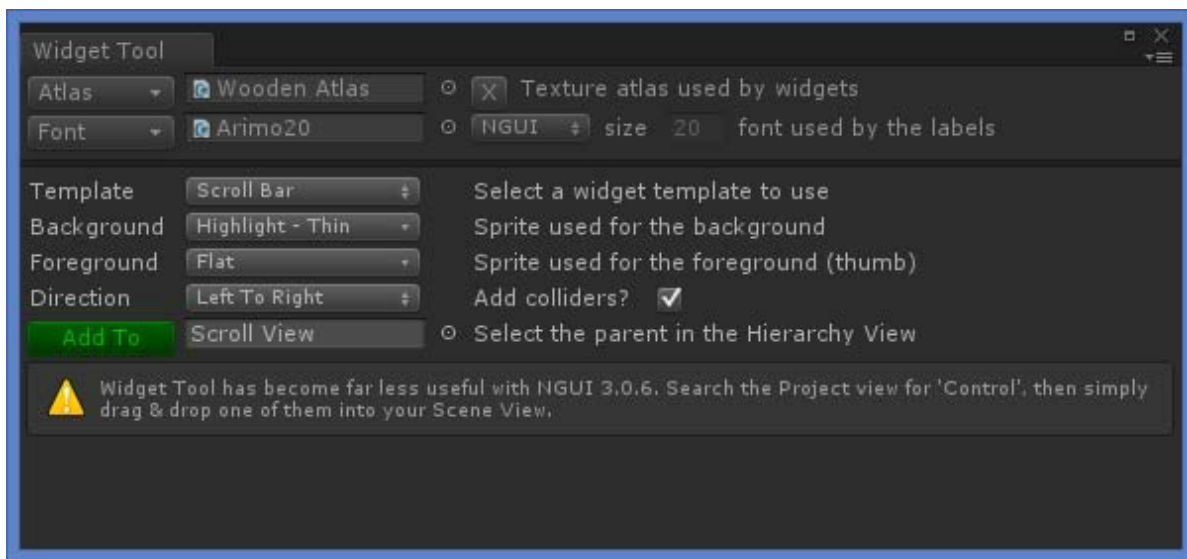
拖动条有多种创建方式，拖动条非常像一个进度条，鉴于拖动条的不同需求，它有非常多的创建方式，我们来简单介绍几种。拖动条一般只有水平方向的拖动条和竖直方向的拖动条两种形态，图3.38展示了一个拖动条的样子，这是一个水平方向的拖动条。



▲ 图3.38

#### 1. 通过WidgetWizard (Legacy) 创建

在Unity顶部NGUI菜单，选择Open选项，打开WidgetWizard (Legacy)，会弹出3.39的界面。



▲ 图3.39

这个界面我们在之前也讲过其用法，它是旧版本的NGUI的控件创建系统。在这个界面中，我们设定好图集和字体（如无需要可以不设定）后，在 **Template** 中选择 **ScrollBar**，然后在 **Background** 中设定它的底板的 **Sprite**，在 **Foreground** 中设定拖动块的 **Sprite**，在 **Direction** 中设定它的滑动方向。然后单击 **Add To** 就可以在相应的UI节点下创建出一个拖动块。

通过这种方式创建的滚动条我们需要调整它的 **Background** 和 **Foreground** 到我们需要的样子。

## 2. 通过PrefabToolBar创建

我们依然在Unity顶部NGUI菜单中选择 **Open**，打开 **Prefab Tool Bar**，弹出NGUI的预设界面，从中将“**Simple Horizontal**”（水平方向的拖动条）或者“**Simple Vertical**”（竖直方向的拖动条）拖动到场景中我们需要的UI节点下，即可完成创建。

通过这种方式创建的滚动条因为是预设体，所以需要调整它的大小和颜色。

## 3. 自己组装一个拖动条

拖动条在旧版本中是用**ScrollBar**组件来实现的，但是，在新版本的NGUI中，可以利用制作进度条的原理来方便地制作它。为了了解它的组件原理，我们还是需要学习一下怎样自我组装一个拖动条出来。

我们观察拖动条具有以下特点：

- (1) 有一个底板槽来显示可以拖动的范围；
- (2) 有一个滑动块，可以在范围内滑动。

通过对比我们可以发现拖动条和进度条的区别：拖动条就是一个缺少表层进度条的可拖动进度条！

所以，我们通过3.6节讲的方法，创建一个可以拖动的进度条即可，在制作这个进度条时，我们需要设置**Thumb**，但是却掉**Foreground**的设置，一个拖动条就制作完成了。

### 3.8.6 拖动条说明

通过不同的方式创建的滑动条有着不同的结构和不同的组件，可能会让我们迷惘，比如说我们通过 **WidgetWizard (Legacy)** 创建的滑动条和通过 **PrefabToolBar** 创建的滑动条功能都一样，但是结构完全不一样、组件也完全不一样。前者依赖于 **ScrollBar** 进行工作，后者依赖于制作进度条使用的核心组件**UISlider**进行工作。在这里我们通过如下说明来理清概念，避免混淆。

(1) **ScrollBar**和**UISlider**我们观察组件界面可以发现，它们非常相似。它俩创建的滑动条从本质上说工作原理是一模一样的，没有任何分别，所以不用去担心哪一种更正规哪一种更好，因为它们都一样。

(2) **ScrollBar**是旧版本的NGUI用来制作滑动条的，在新版本的NGUI中，随着**UISlider**的强大起来，我们可以更方便地使用**UISlider**制作滑动条了，所以推荐使用**UISlider**，也就是制作进度条的方式来制作滑动条。

(3) 我们依然推荐使用自己手动制作的方式去制作拖动条，而不是使用旧的控件创建系统（WidgetWizard）和PrefabToolBar。即使PrefabToolBar是新版NGUI的功能，因为我们每个人的项目中的UI体系设计都不一样，所以熟悉NGUI的控件原理后，推荐自己去拼装控件。

(4) 切记，滑动条就是一个没有Foreground的UISlider。

### 3.8.7 让视图内的内容可以被拖动

我们从上文中已经学会了制作ScrollView和滑动条等知识，但是到目前为止，我们的滑动视图依然没有看到效果，因为现在为止还不知道怎样去拖动内容。如果只是填充一些内容进去，我们会发现，即使有ScrollView也无法滑动里面的内容。这是因为需要对ScrollView所包含的内容进行一些配对设置，以使它能够在ScrollView内被滚动起来。

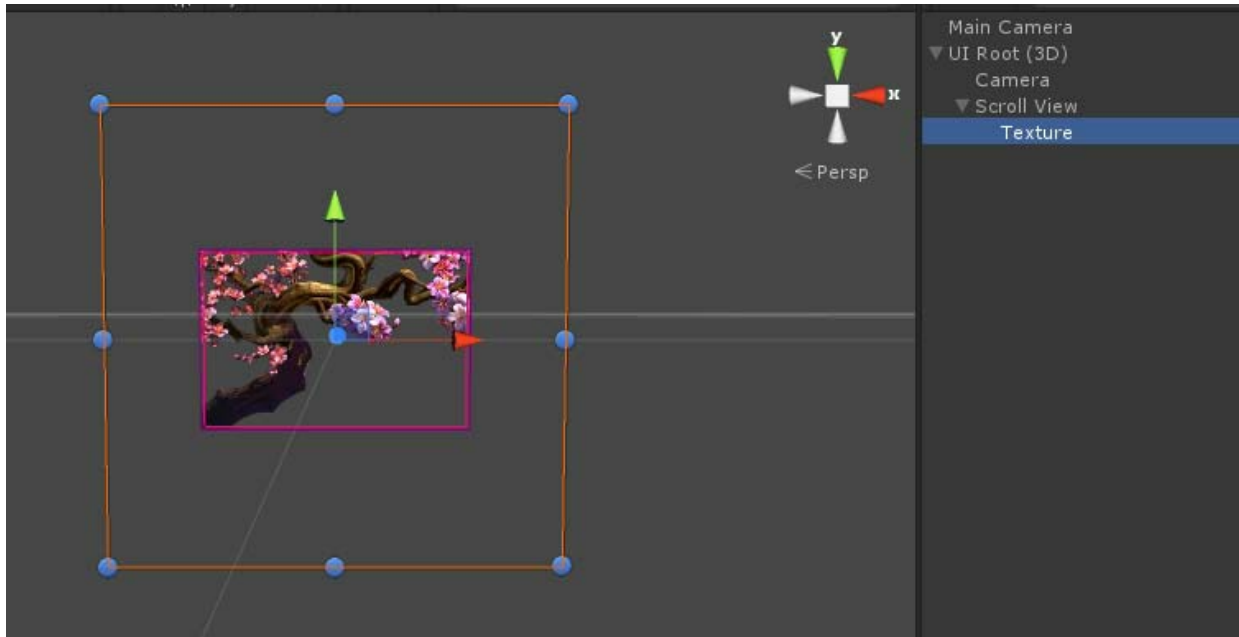
如图3.40所示，我们创建了一个ScrollView，在其下面创建了一个Texture子物体。子物体的Texture我们设置了一张巨大的桃树图片。从图3.40中可以看到，桃树在滚动视图的范围内只能显示一小部分。

下面我们要让桃树可以被滑动以进行浏览，这就是滚动视图的意义所在。

我们选中桃树的Texture物体，首先为其Attach（Unity顶部NGUI菜单选择Attach）一个BoxCollider，因为，既然我们需要拖动这个桃树的图片来查看，就必须让桃树能接收到我们拖动的事件，所以一定得有一个BoxCollider。

然后为桃树的Texture物体附加一个核心组件：Drag ScrollView，附加方式可以在Inspector面板中单击AddCompnent → NGUI → Interaction → Drag ScrollView。添加好组件之后无需进行任何设置，运行游戏时它会自动地去读取父物体中的ScrollView。

然后运行游戏，拖动视窗内的桃树图片，发现已经可以通过拖动桃树图片来进行浏览了。可以拖动的方向是ScrollView中Movement选项中设置的方向。



▲ 图3.40

### 3.8.8 制作滚动视图时的注意事项

滚动视图是一种比较复杂的控件类型，它需要多种控件配合使用，我们在使用时如果忘掉了一些制作的细节，则会导致BUG出现。所以，我们在制作滚动视图时，一定要深刻的理解它的工作原理，除此之外，还要牢记以下一些注意事项：

（1）通常情况下，滚动视图一定要有一个UIPanel来进行窗口剪辑。这个UIPanel组件在创建ScrollView时会自动生成。

（2）拖动条对于滚动视图来说可有可无，没有它滚动视图也能通过移动设备的触屏和PC设备的鼠标光标来进行滚动。

（3）滚动视图内包含的内容，一定要有一个BoxCollider，否则无法接收事件。

(4) 滚动视图内包含的内容，一定要有一个DragScrollView组件，这个组件会和ScrollView相互作用，在运行时，它会自动去找到父物体中的ScrollView，然后和它相互作用，让视图内的内容可以被滚动起来。

(5) 滚动视图的内容，最好放到创建的ScrollView节点下面作为子物体存在，这样可以免去大量的烦恼和隐患。

(6) 滚动视图的滚动方向不要弄错了。

(7) 滚动视图的剪辑窗口的尺寸一定要调整到位，尽量别去调整Clip的Center。

## 3.9 制作复选框 (Toggle)

### 3.9.1 怎样判断是否应当使用复选框

复选框，就是对一个选项做上一个标记，表示这个选项已经被选中了。例如，我们在餐厅点菜时，经常会有服务员递给我们一个菜单，每一项菜品后面都有一个方框，需要点什么菜就在后面的方框内划上一个勾表示这个菜品选中了。在游戏中，复选框一般用来做一些选项的控制，这种选项一般都只有两种答案：是与否。例如，单击一下开启音乐的复选框，这个复选框上就打了一个勾，然后音乐在游戏中就会开启；如果再单击一下，则这个勾会取消掉，然后音乐将会在游戏中关闭。这就是复选框最常见的用法。

当我们要判断是否要使用复选框时，可以遵循以下规律。

(1) 该功能只有两种选择状态：是、否。

(2) 该功能同一时间只能激活且必须激活一种选择状态。



(3) 该功能的两种状态为互斥关系，如果选择了一种状态，则自动关闭另一种状态。

简单地说，复选框其实就是一个开关，我们可以通过单击它来切换打开和关闭。一般开关的应用包括了选项勾选和页签。页签是复选框功能的一个高级应用，我们后文会讲。

根据以上的规律可以发现，游戏中很多地方都运用了复选框。例如，系统设置中，通过复选框功能来设定是否播放背景音乐。在角色界面中，通过复选框来确定是否显示时装，打钩则为显示时装，取消打钩则为不显示时装。

### 3.9.2 创建复选框

创建复选框的方法有很多，我们这里介绍两种方法：自己通过组件来拼装复选框控件和使用预设直接创建。

#### 1. 使用预设直接创建

NGUI提供的预设中有复选框这个控件，可以从Unity顶部NGUI菜单中选择Open，然后打开Prefabs ToolBar，在界面中选中Colored CheckBox 或者Simple CheckBox拖动到我们希望创建的UI节点下即可。

#### 2. 自己拼装复选框控件

(1) 首先选中我们希望创建复选框的UI节点，通过Unity顶部NGUI菜单Creat中选择创建一个Sprite，然后设置它的图片，让这个Sprite作为复选框的底框。

(2) 因为需要这个复选框来接收单击事件，所以为这个底框Sprite制作一个BoxCollider，方法可以使用Unity顶部NGUI菜单中Attach一个BoxCollider。

(3) 下面需要为这个复选框创建一个复选框的核心组件：UIToggle，选中底框后，通过选择



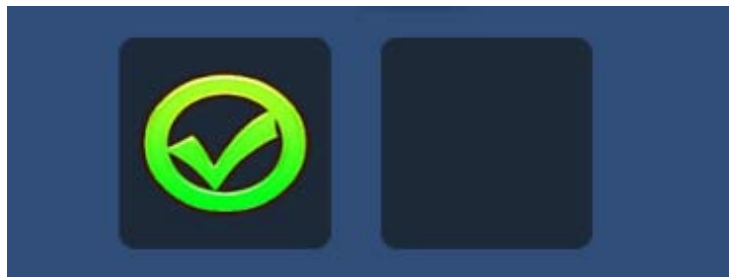
AddComponent → NGUI → Interaction → UIToggle 为这个复选框的底框附加一个UIToggle组件。

(4) 我们在这个底框的Sprite下面创建一个新的、表示选中状态的Sprite，比如一个勾。

(5) 我们将这个表示选中状态的Sprite拖动到底框UIToggle组件中StateTransition模块下的Sprite选项中，然后将底框的UIToggle组件中的StartingState勾上。

到此为止，复选框就算创建完成了，运行一下可以发现默认情况下，它是打钩选中状态，单击一下则勾消失，只剩一个底框；再单击一下又打上了勾，选中了。如图 3.41 所示，左边表示选中状态，右边表示未选中状态。

一般情况下，我们都会复选框的旁边写上一些文字，表示这个复选框代表的什么选项。具体做法也是在底框物体下面创建一个Label作为子物体，然后调整位置即可。



▲ 图3.41

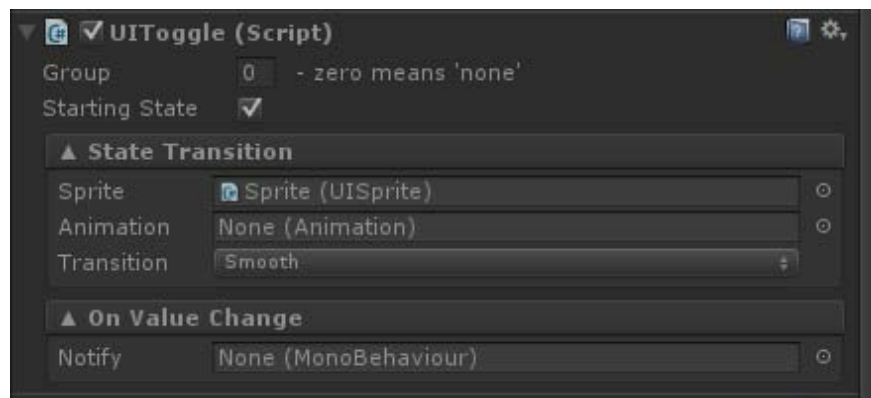
### 3.9.3 复选框的核心组件UIToggle

在制作复选框时，已经发现它的核心组件除了接收事件的BoxCollider 以外，就是一个我们新认识的 UIToggle 组件。这个组件的主要作用是一个开关，通过它在打开和关闭之间进行切换。

UIToggle组件的设置如图3.42所示，我们先来了解一下它的设置。

#### 1. Group

开关组的设置。默认为0，表示没有开关组。



▲ 图3.42

当有多个Toggle的Group相等且都不为0的时候，表示它们在同一个开关组当中，同一个开关组内的开关只允许打开一个。例如，我们有开关A、B、C，它们的Group都为1，当打开A的时候，B和C会自动关闭；同理，当打开B的时候，A和C会关闭。

当一个Toggle不属于某一个开关组，既Group为0时，它就属于一个独立开关，可以通过自身来进行开关，当它属于某一个开关组时，就无法通过单击自身来进行开启和关闭了，因为一个开关组内必须要有一个开关是处于开启状态，它必须通过开关组内部的多个Toggle之间的切换来进行关闭和开启，例如，刚才举的A、B、C3个开关的例子，如果A是打开的，我们希望关闭A，则我们需要打开B或者C才能使A关闭，因为A已经不是一个独立开关，而是ABC开关组的一部分。

开关组的使用原理和一个非常常用的UI控件功能很像：页签。页签也有着类似的功能需求，有很多个页签，我们同一时间内只能查看一个页签，如果选择了一个页签，则之前打开的页签自动关闭。

页签是Toggle的一个高级应用，我们这里暂时只讲复选框的制作。

## 2. Starting State

是否初始状态，如果选中，则为初始状态，不选中则不是。

这个设置项的意义在于：当开关是一个独立开关（Group 为0）时，勾选Starting State意味着在初始状态下，开关属于开启状态。当开关属于一个开关组时，勾选Starting State则意味着在初始状态下，这个开关组中初始处于打开状态的为这一个开关。

当一个开关组中有一个以上的开关都勾选了Starting State时，则以这个组中排在最后的一个勾选了Starting State的Toggle为默认开启的开关。

### 3. State Transition模块

这个模块是为了设置勾选的时候的一些关联UI表现。

- Sprite** 是设置选中状态下要显示出来的 Sprite，也就是之前创建复选框时创建的那个用来表示选中的打勾的Sprite。设置的方法为，将表示选中的Sprite拖动到这个选项中即可。

- Animation** 是设置状态切换时的动画，例如，需要选中时会会有一个勾从底框中蹦出来弹两下，那么就可以制作一个蹦出来弹两下的动画拖动到这个选项中。

- Transition** 是选择开关切换时的一个平滑效果，里面提供了两种选项进行选择：**Smooth**和**Instant**，如果我们选择**Smooth**，则在进行开关切换时，表示选中的那个Sprite的消失和出现会显得更加平滑一些；如果选择**Instant**，则开关切换时，表示选中的那个Sprite的消失和出现会瞬间消失和瞬间出现。这个选项默认的设置**Smooth**，一般情况下**Smooth**的用户体验更舒服一些，所以，一般情况下我们不需要去调整它。

### 4. On Value Change

这里是设置当开关状态改变时触发的函数，设置方法和其他控件一样，就不再详细描述了。

## 3.10 制作下拉菜单 (PopupList)

### 3.10.1 怎样判断是否应当使用下拉菜单

下拉菜单，就是将一系列的选项隐藏，通过单击某一个控件将会弹出一个包含这些选项的列表，在其中选择想要的选项。这样做不但可以节省屏幕空间，也可以让用户在进行选择时更加方便快捷。

下拉菜单本质上还是一个单选框，与 **Toggle** 的功能有一些类似，对于下拉菜单玩家必须选择一个选项（有一个默认的初始选项），在同一时间也只能选择一个选项（单选性质）。在游戏开发过程中，如果碰到了以下特点的需求，就可以考虑使用下拉菜单了。

- (1) 有一系列选项需要玩家做出选择，这些选项是有限多的。
- (2) 这些选项玩家必须选择一个，也只能选择一个。
- (3) 这些选项如果全部列出来用**Toggle**制作单选功能会非常占用屏幕空间。

如图3.43所示为一个典型的下拉菜单。

### 3.10.2 创建下拉菜单

我们可以有多种方式来创建下拉菜单，这里主要学习两种方式。

#### 1. 第一种方法：使用NGUI做好的下拉菜单预设体

在Unity顶部NGUI菜单，依次选择Open → Prefab ToolBar，打开了NGUI内置的预设工具界面，在其中选择Simple PopupList拖动到希望创建的UI节点下，就算创建完成了。但是，这种方法含有很多预定义的东西，如果对PopupList的原理非常熟练，那么不建议使用这种方法。



▲ 图3.43

## 2. 第二种方法：自我拼装一个

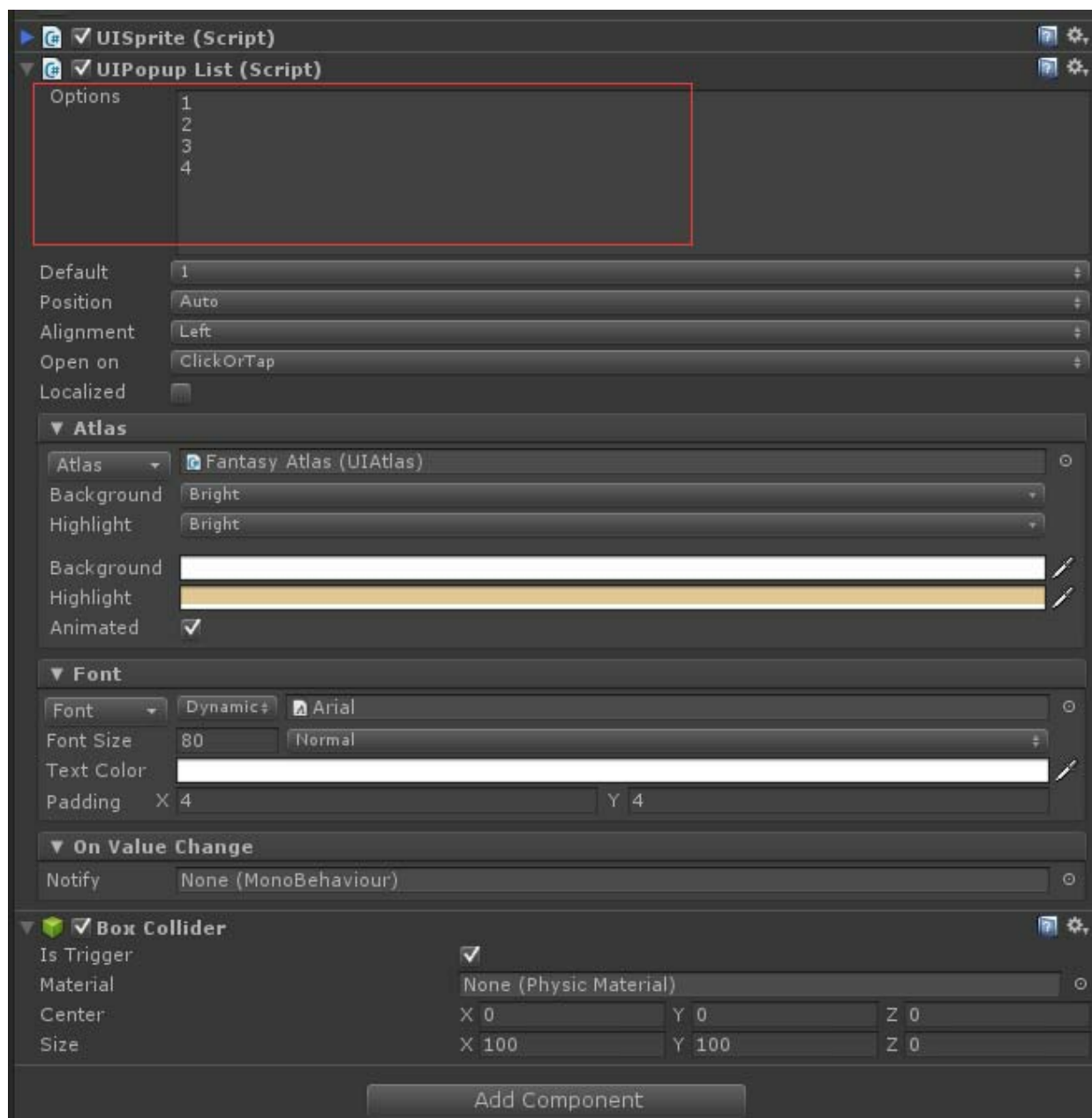
首先选中希望创建下拉菜单的 UI 节点，在 Unity 顶部菜单，依次选择 **Creat** → **Sprite**，这样就在UI节点下创建了一个**Sprite**子物体。

然后在这个子物体身上添加一个**PopupList**组件，添加方式为，在 **Inspector** 面板中依次单击

**AddComponent** → **NGUI** → **Interaction** → **PopupList**。

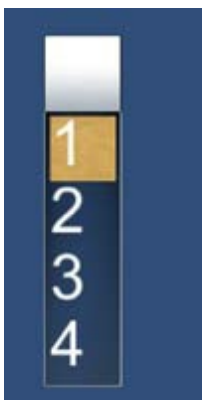
因为下拉菜单需要单击，所以，还需要为它添加一个**BoxCollider**组件，添加方法为选中这个控件，在Unity顶部NGUI菜单，依次选择 **Attach** → **BoxCollider**。

这样就算创建完成了，组件截图如图3.44所示。



▲ 图3.44

可以在图3.44中红框所示部分输入一些选项，每输入完一个选项，就必须回车换行再进行输入下一行，因为NGUI会按行数来识别选项。我们在图3.44中输入了1、2、3、4 4个选项，运行游戏后，单击这个PopupList物体，可以看到弹出了一个下拉菜单，如图3.45所示。



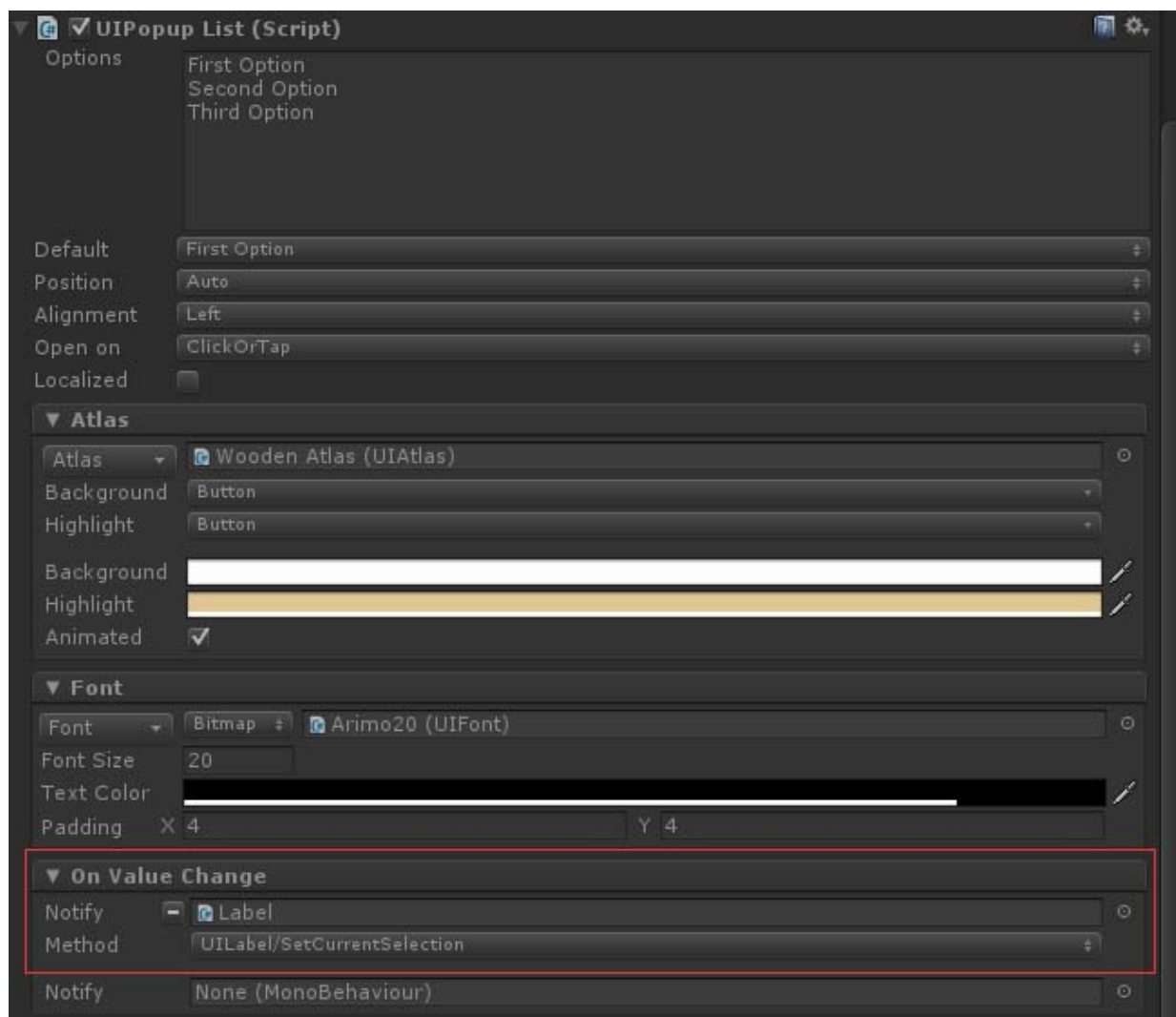
▲ 图3.45

### 3.10.3 显示当前选中的选项

在图3.44中可以看到我们制作的下拉菜单并没有显示当前选中的选项，如果需要显示当前选中的选项，则可以在这个下拉菜单的控件下创建一个Label子物体，创建方法为选中这个下拉菜单的物体，在Unity顶部NGUI菜单，依次选择Creat → Label。

然后将这个 Label 和下拉菜单关联起来，我们将 Label 拖动到下拉菜单 PopupList组件的On Value Change 回调中，选择 SetCurrentSelection 方法，这样当PopupList 的选项被改变时，当前被选中的选项会实时更新到这个被关联的Label上显示出来，如图3.46中红框部分所示。

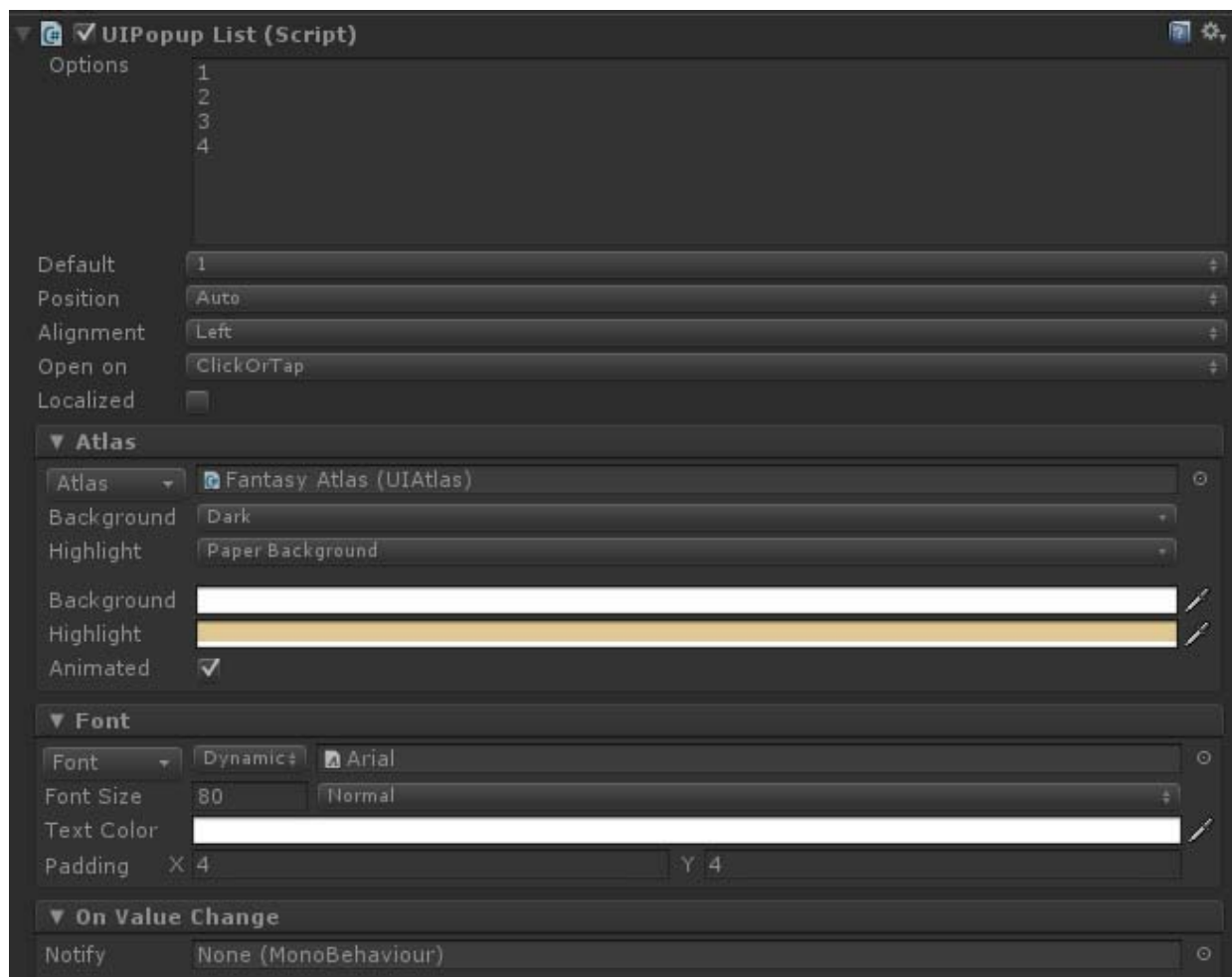




▲ 图3.46

### 3.10.4 下拉菜单核心组件PopupList

制作下拉菜单需要一个Sprite来被单击，需要一个BoxCollider来接收单击事件，但是，最核心的功能几乎全部依赖于一个核心组件：PopupList。这个组件的界面如图3.47所示。



▲ 图3.47

### ●Options

这里是下拉菜单的各个选项录入的地方，识别方式为按行识别，也就是说每填入一个选项后，需要回车换行才能继续录入下一个选项。

### ●Default

默认情况下选中的选项，这个选项会自动填充为我们录入的第一个选项。

### ●Position

位置，这里给了3个选项。

**Auto:** 菜单将会自动决定是从上方弹出还是下方弹出。

**Above:** 菜单将会从上方弹出。

**Blow:** 菜单将会从下方弹出。

### ●Alignment

对齐方式，这里和Label里的对齐方式一样，就不多赘述了。

### ●Open on

打开的方式，这里提供了以下的方式可选择。

**ClickOrTap:** 单击出现菜单。

**RightClick:** 右键单击出现菜单。

**DoubleClick:** 双击出现菜单。

**Manual:** 手动出现，这种模式下任何输入都不会出现，必须代码控制它出现。

### ●Localized

这里是指菜单中的文本是否被本地化。

所谓本地化，可以理解为多语言翻译，例如，我们的游戏是英文的，那么游戏中的所有UI文本都将会被转换为英文版本。那么这个**Localized**选项就是决定这个下拉菜单是否被转换的。

如果打上勾，则表示这个菜单里的选项也会被本地化转换语言。如果不打钩，则表示这些选项里的文本不会被翻译，永远会保持它本来的样子。

为什么要有这么个选项呢？比如我们在选择语言的下拉菜单里，里面的每一种语言选项不能被翻译，因为玩家要依赖里面它认识的选项去选择自己要的语言。如果这里的选项被翻译成同一种语言了，那么就没有选择语言的意义了。

### ●Atlas

图集设定，这里有很多设置项。

**Atlas:** 选择图集。

**Background:** 设定下拉菜单的背景的精灵图片，还可以设置颜色。

**Highlight:** 设定下拉菜单出现后，鼠标光标移到选项上高亮显示时显示的图片，也可以设定颜色。

- Font**

设定菜单文本的字体、字号大小等。

- On Value Change**

当这个下拉菜单当前选中选项变化时，触发的事件。

可能我们会有疑问：为什么这个下拉菜单会单独进行是否本地化和使用 **Font** 字体的指定呢？这是因为NGUI考虑到大多数游戏在制作上有语言版本时，都会使用下拉菜单来让玩家选择它所认识的语言，这种情况下选项不能被翻译，使用的字体也可能很特殊要涵盖很多种语言文字，所以，**NGUI**是一个非常贴心好用的插件。

### **3.10.5 制作下拉菜单的注意事项**

我们在制作下拉菜单时，一定要注意以下事项：

- (1) 一定要有接收单击事件的**BoxCollider**;
- (2) 填写选项时，一定要注意换行;
- (3) 如果制作下拉菜单是为了让玩家选择语言，则要更加注意本地化的设置和**Font**的设置。

# 第4章 UI动画

## 4.1 常见的两种UI动画介绍

### 4.1.1 要区分UI动画和UI特效两个概念

UI动画的目的是为了让UI能够动起来，它的本质原理是通过每隔一定时间去改变UI的某个参数来实现动画效果。例如，每隔0.5秒就让UI的Transform组件的Position的X轴正向移动1个单位，持续5秒，那么就构成了一个5秒内UI朝X轴正向运动的动画。也就是说，UI动画是在改变UI的组件参数，让UI实实在在地动起来了。

而UI特效，虽然效果上看着是UI特别绚丽的动态效果，但是，它的本质是叠加了一个特效在UI上，UI本身是没有发生任何变化的。例如，我们经常在游戏中看到领取奖励的UI按钮在可以领取奖励时，会不停地闪烁发光，这其实是在领奖按钮上叠加了一层闪烁发光的特效，当可以领奖时，这个特效就激活。

所以，UI动画和UI特效是两个概念，UI特效具有独立性，可以通过叠加特效让UI看上去很绚丽，但是无法让UI本身动起来。而UI动画可以让UI本身动起来，却无法给UI赋予额外的特效效果。UI特效需要额外的特效资源，UI动画不需要任何的额外资源。

### 4.1.2 关于Tween动画

**Tween**动画是动画中一种非常常用，同时也非常简单的一种动画。**Tween**动画简单一点的解释是：中间动画。它的原理是设定动画的起点和终点，以及这过程中每一个中间点的值，然后让物体按照设定的这个流程去插值地改变值。

例如，我们通过**Tween**动画来做一个物体的X轴位置改变的动画，设定该物体X轴位置的**Tween**路线为0、10、5这3个点，设定好了之后，物体的X轴位置就会变化为0，然后很平滑地变到1、2、3.....10，这个过程会根据时间计算出它变化的速度，然后一点一点地去改变，而不会瞬间变为10，这就是所谓的插值。当物体的X轴位置变为10之后，它又会很平滑地慢慢变为5，然后动画结束。

利用**Tween**动画可以做很多UI的动画效果，比如按钮被单击时会弹一下、按钮在某种情况下开始自己一蹦一蹦地上下浮动等。

在NGUI中，自带了一套非常精简实用的迷你**Tween** 动画库，这个动画库虽然不能支持强大的**Tween**动画效果，但是，在UI的动画表现中已经非常实用。我们将会在后文详细讲解这个迷你**Tween**动画库。

### **4.1.3 关于Animation动画**

**Animation**动画就是Unity引擎自身的**Animation**动画系统，它的功能非常强大，从原理上来说，它和 **Tween** 动画几乎是一样的原理，通过插值去平滑地过渡每一个关键帧。但是，它支持Unity引擎中绝大部分组件的绝大部分参数，而且支持各种各样的运动曲线编辑，可以让动画按照一个自定义的、任意的节奏进行变换，类似于**Tween**动画的终极形态。

在NGUI中自带的迷你**Tween** 动画库有一定的局限性，它只能实现一些常用的简单的动画效果，而**Animation**是Unity引擎原生的动画系统，非常强大。例如，我们希望能让一个点光源的灯光不停地明暗变

化，做出一种夜晚灯光闪烁的效果，或者天上星星闪烁的效果，我们使用NGUI自带的Tween动画库就难以做到，就要求助于一些专业的Tween动画库，但是，我们也可以使用Animation轻松做到类似效果。

使用Animation来制作UI的动画主要是对付一些复杂的动画效果，一般情况下，优先使用NGUI自带的Tween动画，因为它简单，利于操作和维护。在碰到特殊情况下，才去使用Unity自身的Animation动画系统。

下面我们将逐一介绍NGUI的迷你Tween动画库中的一些常用的动画效果。

## 4.2 渐隐渐现动画（透明度动画）

### 4.2.1 透明度动画的介绍和应用

透明度动画的原理是改变UI控件的Alpha值来实现。在前文我们讲解UI控件的时候，讲到了很多UI控件都带有Widget的设置，如Sprite、Label等都有Widget的设置，在Widget设置里可以改变控件的颜色（其中包括了透明度）、尺寸、深度等。透明度动画改变的Alpha值就是改变的这里的颜色设置中的Alpha值。

使用透明度动画可以做UI的渐隐和渐现，比如一个UI按钮出现时，它是慢慢地从透明变为不透明出现，消失时也是慢慢地透明掉。

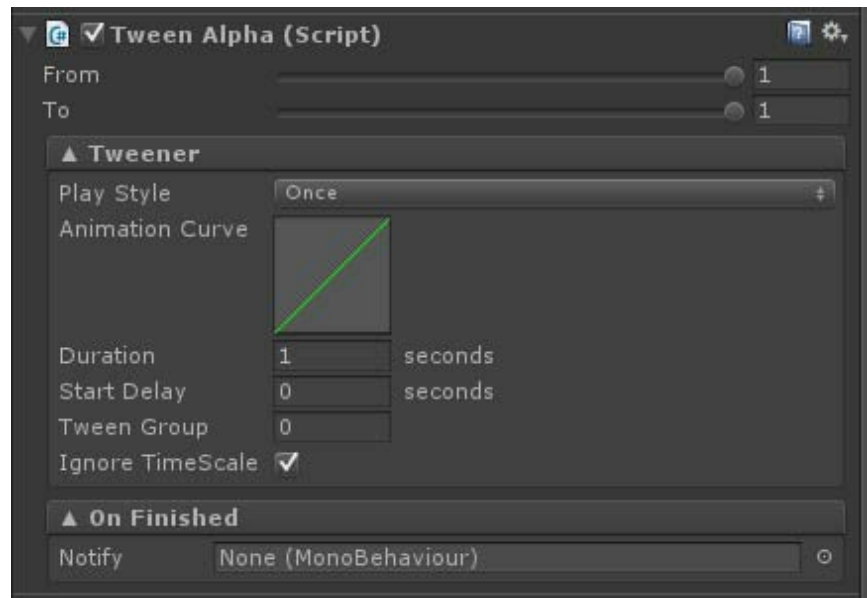
我们也可以使用透明度动画做UI的若隐若现效果，例如，界面中有一行文字时而消失时而出现，不停地重复这个动画过程构成了一个若隐若现的效果。

### 4.2.2 使用透明度动画TweenAlpha



我们需要为某个 UI 增加透明度动画时，可以通过为它增加一个 TweenAlpha 组件，这个组件在 NGUI 的 Tween 动画库中，添加方式为依次选择 AddComponent → NGUI → Tween → TweenAlpha。

添加成功后的 TweenAlpha 组件设置界面如图 4.1 所示，我们来了解一下如何设置它来制作一个透明度动画。



▲ 图4.1

### 1. From和To

这是 Tween 动画的核心设置项，起始点的设定。值得注意的是，在 NGUI 自带的迷你 Tween 动画库中，一个 Tween 动画一般只支持起始点的插值动画。

在 TweenAlpha 中，我们可以在 From 和 To 中设置动画的初始透明度和结束时的透明度，0 为全透明。1 为默认值，也就是不透明。

### 2. Play Style

这里是播放的风格，可以理解为循环模式，一共有以下 3 种形式。

- Once

只播放一次，播放完了之后，就停止。

- Loop

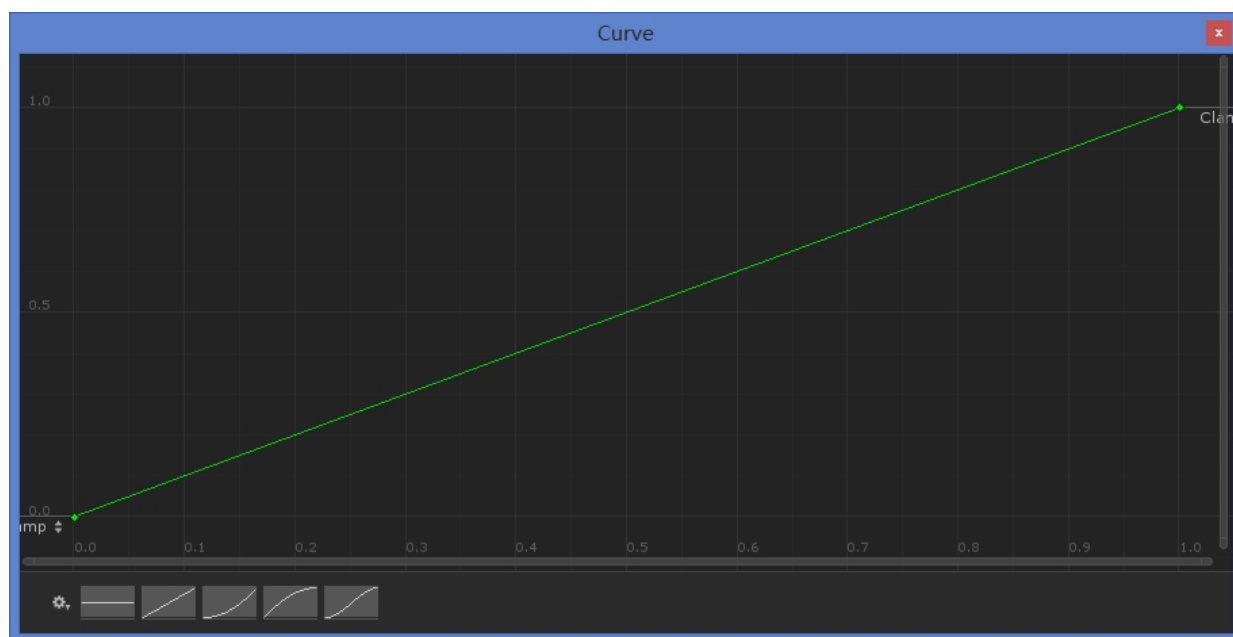
循环播放，播放完了之后它会立即瞬间回到起点接着播放。我们举一个比较形象的例子：有一个动画是从透明度为1变为透明度为0，那么Loop模式下，动画播放时透明度的变化为1→（中间插值）→0→瞬间变为1→（中间插值）→0→..... 如此无限循环下去。注意：这里从0慢慢插值达到终点1之后，是瞬间变回0进行下一遍播放的。

### ●PingPong

乒乓模式，顾名思义，动画会像乒乓球一样来回播放，它和Loop最大的区别在于，Loop在播放完了之后是瞬间回到起点然后继续播放，而PingPong模式在动画播放完了之后，会倒着插值播放。还是那个例子：有一个动画是从透明度为1变为透明度为0，在PingPong模式下，动画播放时透明度的变化为：1→（中间插值）→0→（中间插值）→1→（中间插值）→0→..... 如此无限循环下去。

## 3. Animation Curve

这是动画的曲线编辑，可以编辑一些动画的节奏。我们单击这个设置项中的曲线图片，会弹出以下界面，如图4.2所示。

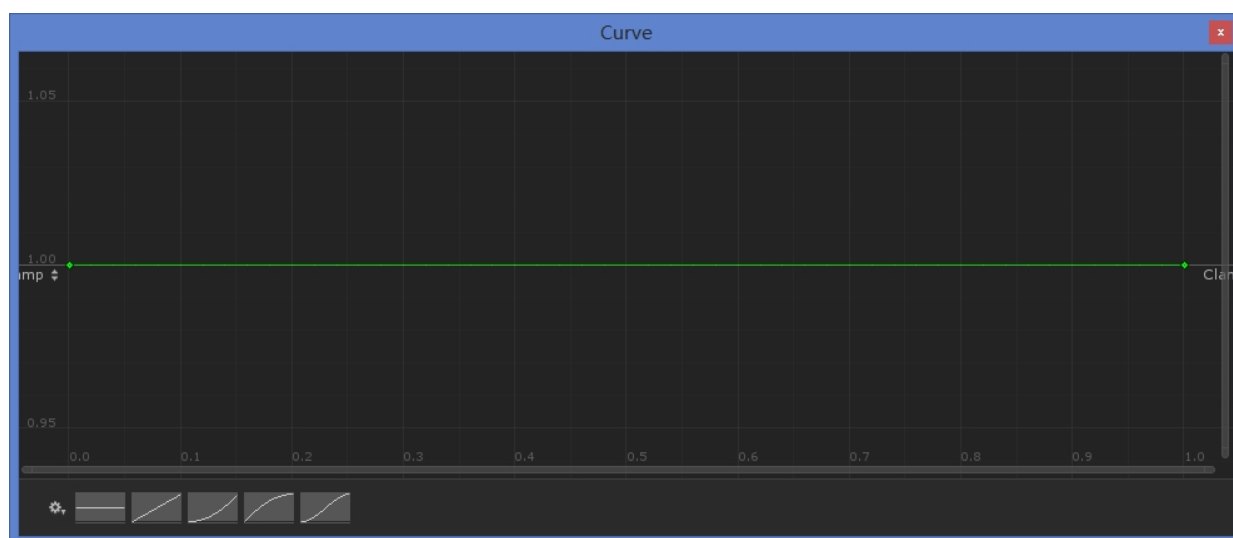


▲图4.2

从图4.2中可以看到一条默认的曲线，为一条直线。它是单次动画内的曲线编辑，直线的意思是起点的值会在持续时间内，非常平均地进行插值改变，最后变为终点。我们在图4.2中底部的几个小灰块中可以看到，有几条预设好的曲线供我们选择，熟悉函数的应该都明白它的意思，我们只需要单击其中一种曲线，即可将这条预设好的曲线作为我们需要应用的曲线。

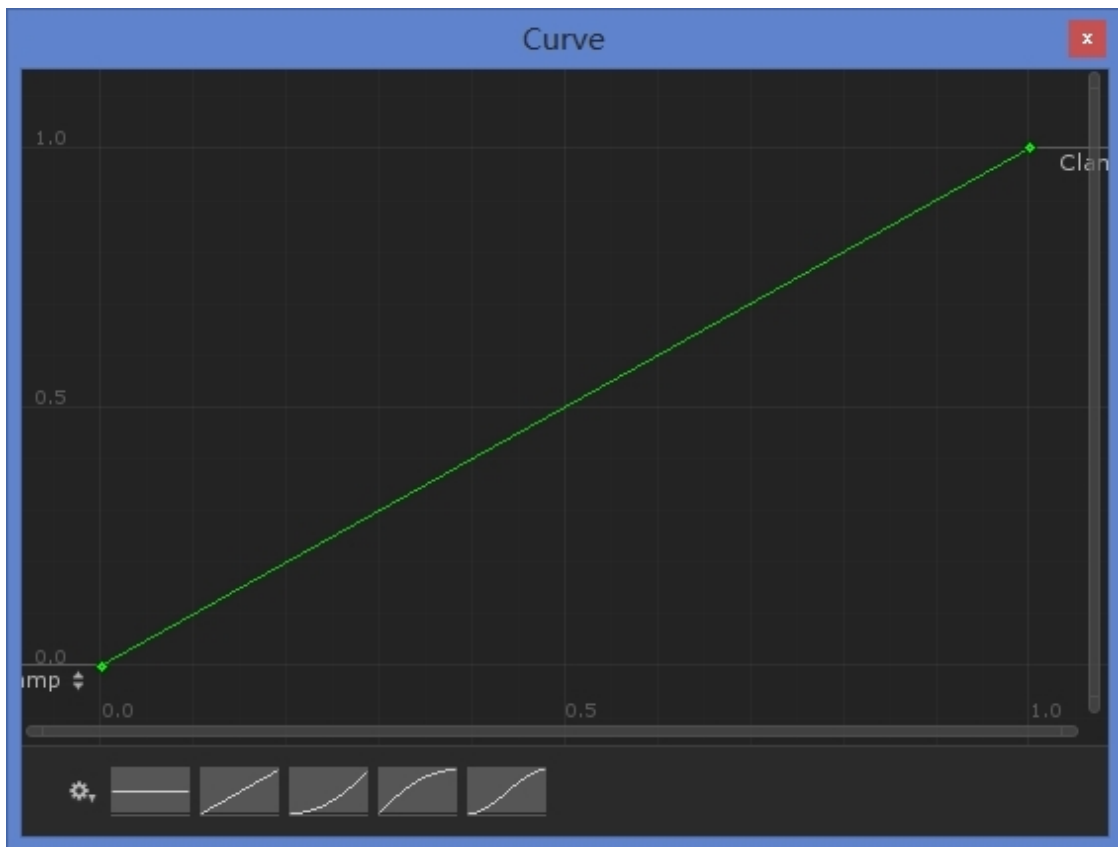
我们简单介绍一下几种预设好的曲线。

如图4.3所示，为水平的一条横线，这种模式比较特殊，代表着没有插值动画，动画在过渡时，不论动画需要过渡多少时间，它都会一瞬间跳到终点，完全没有中间过程，动画一旦开始播放就会立即达到终点播放完成。



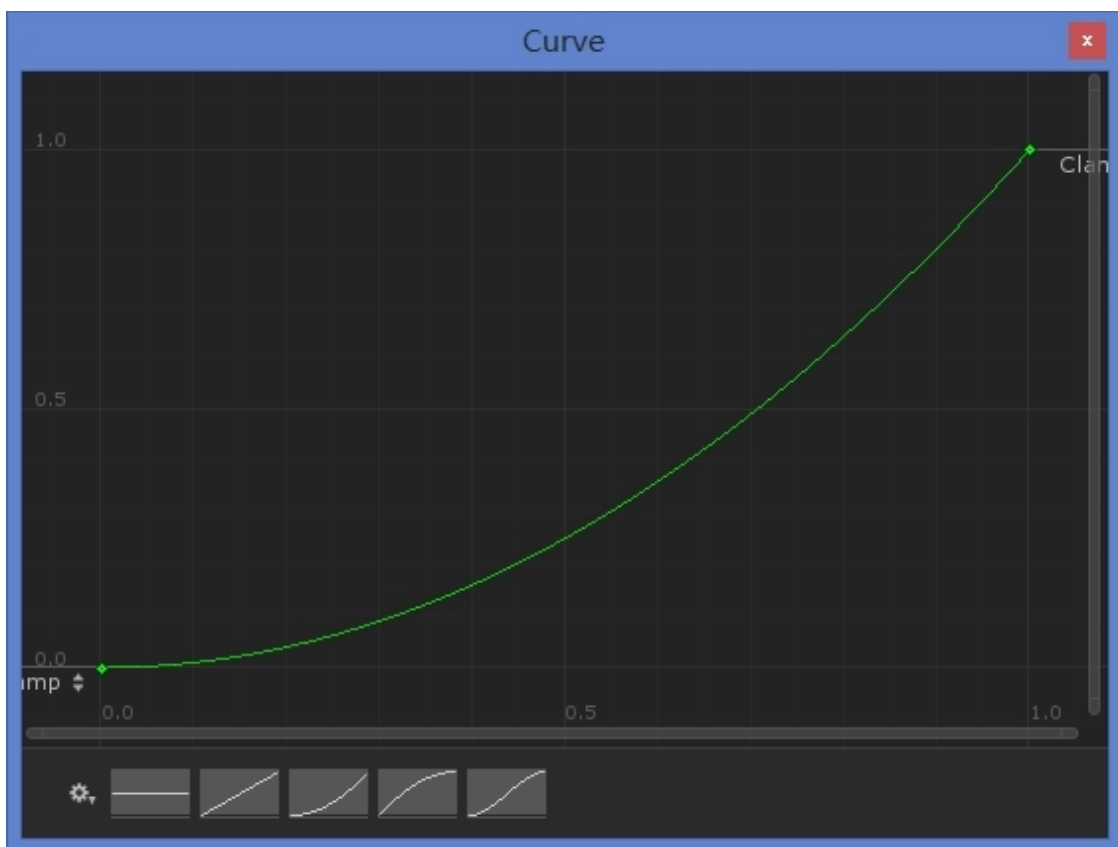
▲ 图4.3

如图4.4所示，为直线，动画将会匀速从起点过渡到终点。



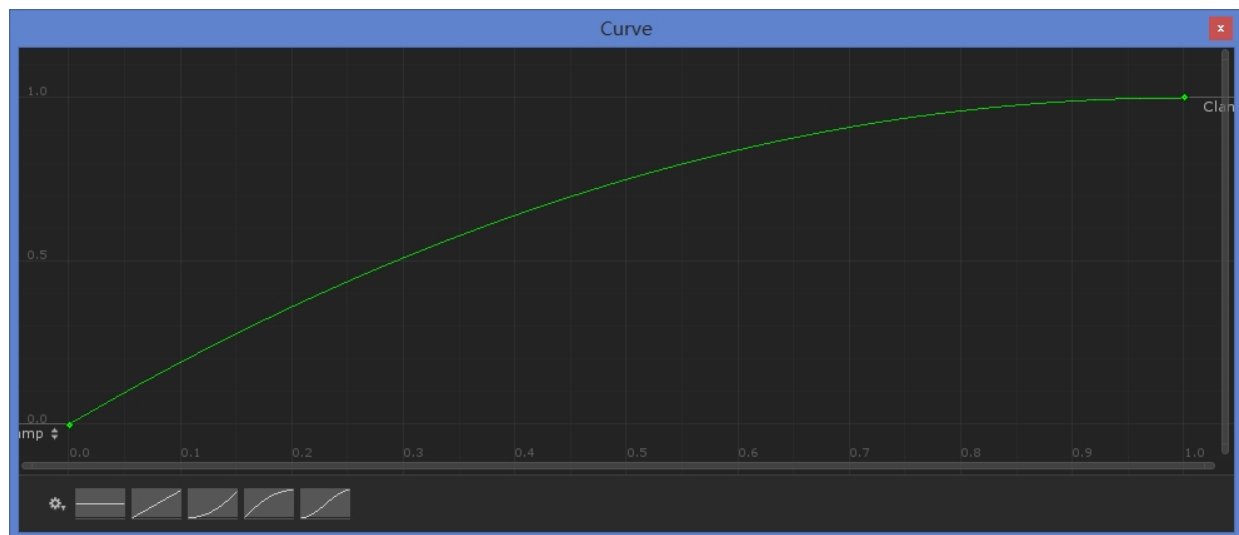
▲ 图4.4

如图4.5所示，为先慢后快的曲线，动画将会以先慢后快的方式从起点过渡到终点。



▲ 图4.5

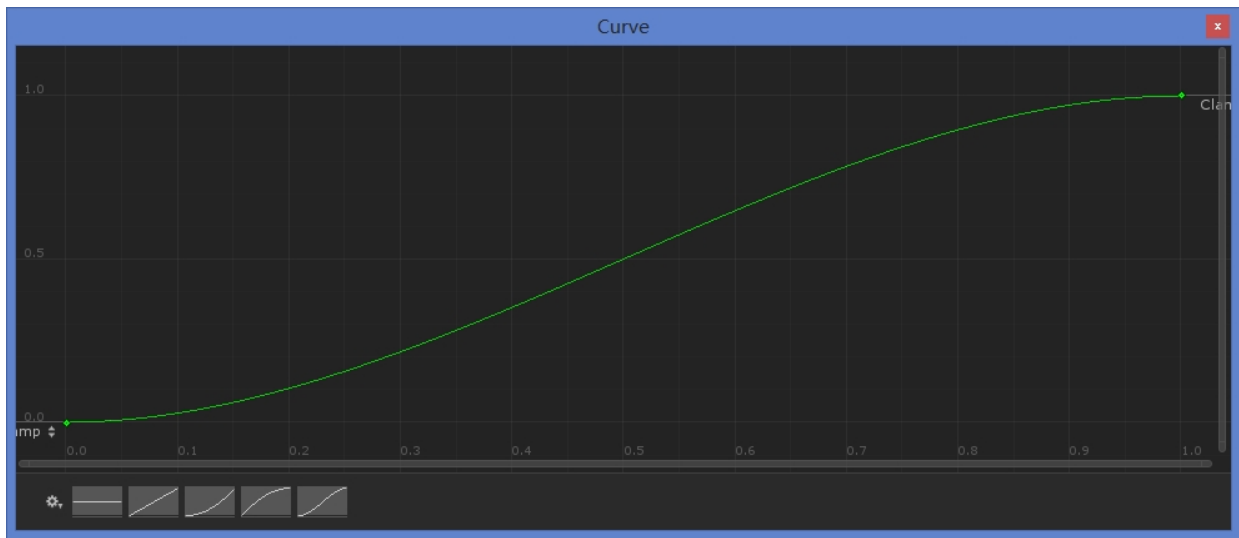
如图4.6所示，为先快后慢的曲线，动画将会以先快后慢的方式从起点过渡到终点。



▲ 图4.6

如图4.7所示，为先慢后快最后再慢的方式，动画过渡时，在开始时比较慢，然后变快，快到达终点时又开始变慢。

如果我们对它预设的曲线还不满意，我们可以在曲线中进行拖动设置，方法和 Unity 的Animation曲线一样（这个属于Unity的原生系统我们就不多讲述了）。我们也可以通过单击预设曲线左边的齿轮符号，在弹出的预设曲线界面中单击New来新创建一个预设曲线，以便于以后我们更方便地应用自定义的曲线。



▲ 图4.7

#### 4. Duration

持续时间，单位为秒，默认为1秒。这里可以填浮点数，如0.5秒。

#### 5. Start Delay

播放延迟时间，单位为秒，默认为0秒，支持浮点数。

#### 6. Tween Group

动画所属的组，默认为0。这个设置会在后文讲解UIPlayTween时讲到，它的主要作用是通过分组来整体控制多个Tween动画。

#### 7. Ignore TimeScale

是否忽略时间缩放，默认为是。

这个选项用得极少，但是知识点很重要。我们都知道Unity中有一个TimeScale的概念，我们在某些比较简单的情况下会直接使用TimeScale=0这种方式来使游戏中的时间停下，达到游戏暂停的效果。而这里的是否忽略时间缩放，就是指当游戏中的TimeScale改变时，这个动画的时间是否跟着受影响。

## 8. On Finished

在动画播放完毕时候触发的函数，这里的设置和之前UI控件的回调函数设置方法一样，就不多描述了。

### 4.2.3 使用透明度动画的注意点

NGUI的Tween动画看上去都比较简单，容易操作。但是，在实际开发中使用透明度动画时，我们要注意一些比较隐藏的、又比较重要的规律。

(1) 从起点播放到终点，就算动画播放完了一遍。但是，在动画结束的触发事件中，我们需要注意的是，如果动画模式是Loop或者PingPong，那么它将永远不会结束。

(2) 透明度动画会实实在在地改变UI的透明度，并不是一个“临时”透明度。例如，我们设置了一个透明度从1变为0的渐渐消失动画，当透明度变化到0.5时，我们就将动画组件关闭，此时UI的透明度将会一直停留在0.5。

(3) 如果有自定义动画曲线，那一定要检查曲线是否和自己想要的效果一致。

(4) 动画组件激活后，它会立即开启StartDelay的计时，然后播放动画。

(5) 如果动画播放设定为播放一次，那么动画播放一次之后，就会自动关闭该组件。



## 4.3 颜色变化动画（变色动画）

### 4.3.1 变色动画的介绍和应用

变色动画是通过NGUI的TweenColor组件来实时改变UI控件的颜色来实现动画效果。我们之前讲过UI控件的widget中改变Color的设置中，原理是颜色的相乘。UI图片中的每个像素都有一个RGB值来表示这个像素的颜色，在Unity中会将RGB3个值由0~255的数值转变为0~1的一个数，然后和UI控件的widget的Color中设定的颜色值（设置的时候是为RGB分别填入0~255的数，但是也会被转换为0~1的数）进行相乘。在变色动画中，原理也是色值相乘。

例如，设置动画的起点是白色，终点是黑色，那么UI的最终颜色将会是：UI图片的原色\*白色→UI图片的原色\*黑色。这个动画变化过程也是插值的。

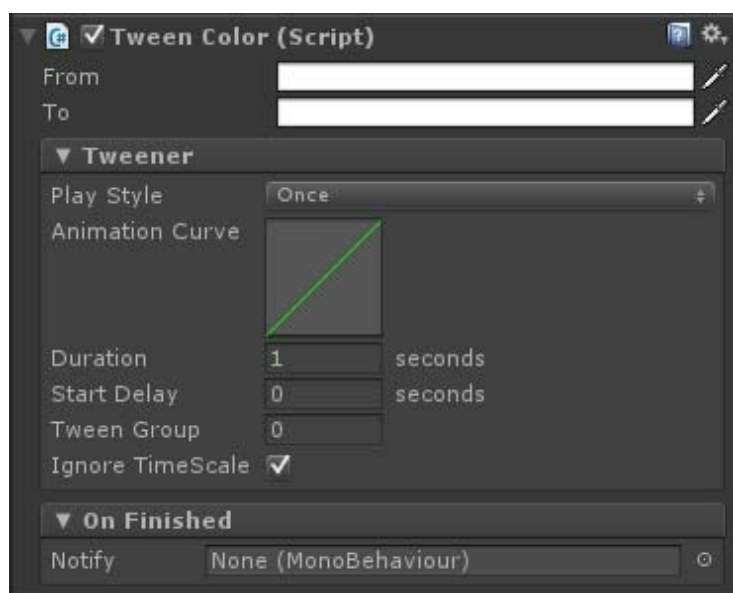
变色动画在游戏中的应用非常广泛，比如我们选中某个UI时，它会明暗闪烁，则是一个白色到黑色的PingPong动画。再比如当某个UI我们无法单击时，它会不停地闪红色，则是一个白色到红色的PingPong动画。

值得注意的是，我们在制作游戏的UI动画时，一定要区分透明度动画和颜色动画，比如说明暗变化，如果用透明度动画来制作，它其实没有变暗，是变得半透明了看上去有一点变暗的效果，而用颜色动画，则透明度没有变化，是颜色实实在在变暗了。

### 4.3.2 使用颜色动画TweenColor

我们需要为某个UI增加变色动画时，可以通过为它增加一个TweenColor组件，这个组件在NGUI的Tween动画库中，添加方式为，依次选择 AddCompent → NGUI → Tween → TweenColor。

添加成功后的TweenColor组件设置界面如图4.8所示，我们来了解一下如何设置它来制作一个变色动画。



▲ 图4.8

从图4.8中可以看到，变色动画的组件和透明度动画的组件从界面上来看非常相似。是的，所有的NGUI自带的Tween动画，几乎都是一模一样的结构，只是其中的设置有一些不同。

TweenColor 一样只支持起点和终点两个点的设置，然后通过插值曲线来过渡。其中Tweening 模块和之前讲过的透明度动画的 Tweening 模块是一样的，这里就不多赘述。如果对Tweening模块还不了解，可以参看4.2节。

关于这个TweenColor组件，我们要着重讲一下它的起始点设置。它的起始点设置是两个颜色设置板，和UI控件的widget模块中的颜色设置板一模一样。我们可以点进去设置想要的颜色。既然是用UI控件的widget模块中的颜色设置板来设置颜色，颜色有RGBA4个值，所以也可以通过设置它的A值来实现一个透明度动画。

### [4.3.3 使用颜色动画的注意事项](#)

使用颜色动画的时候，需要注意以下一些注意点。

(1) 从起点播放到终点，就算动画播放完了一遍。但是，在动画结束的触发事件中，需要注意的是，如果动画模式是Loop或者PingPong，那么它将永远不会结束。

(2) 颜色动画是实实在在地改变了 UI 控件的颜色，并不是“临时改变”。例如，一个动画是将 UI 从白变黑，那么当动画播放完毕之后，UI是真的变黑了，并不会自动变回原来的颜色。

(3) 颜色改变的原理是将 UI 控件的原色和动画中设置的颜色进行相乘，这个原理一定要深刻理解，如图4.9所示，则是一个动画的范例，在TweenColor中设置动画的起点为白色，终点为红色。



▲ 图4.9

## 4.4 位置变换动画（位移动画）

### 4.4.1 位移动画的介绍和应用

NGUI中的每一个UI控件其实都是一个带有NGUI脚本的Unity游戏物体（GameObject），它们都有一个Transform组件来管理这个物体的位置、旋转、缩放。位置变换动画，就是通过改变这个UI物体的Transform组件的Position实现的。

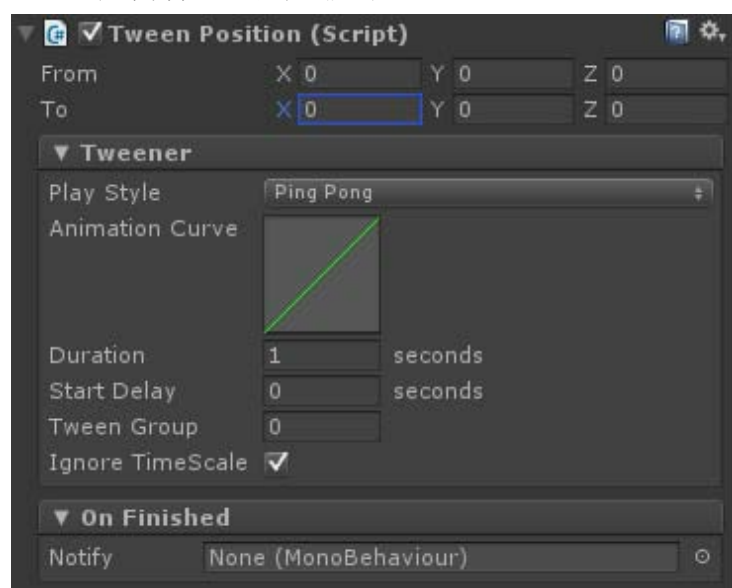
值得注意的是，在NGUI的UIRoot树状结构中的物体，Transform的Position一般都是本地坐标（经过了UIRoot的缩放），而位移动画的核心组件TweenPosition改变的是Position中显示的坐标。也就是说，如果我们对UIRoot下的UI物体使用TweenPosition，则单位可以理解为像素，如果我们自己在摄像机中挂上一个UICamera，然后对这个Camera照射到的物体（不属于某个UIRoot）使用TweenPosition，则单位是Unity中的标准单位：米。

位移动画也是在开发游戏时经常使用的一种动画，比如某个按钮或者图片一直不停地上下漂浮等。再比如我们制作比较常用的“抽屉界面”都会用到位移动画，“抽屉界面”的制作我们后面会讲解。

### 4.4.2 使用位移动画TweenPosition

我们需要为某个UI增加位移动画时，可以通过为它增加一个TweenPosition组件，这个组件在NGUI的Tween动画库中，添加方式为，依次选择AddCompent→NGUI→Tween→TweenPosition。

添加成功后的TweenPosition组件设置界面如图4.10所示，我们了解一下如何设置它来制作一个位移动画。



### ▲图4.10

TweenPosition的组件设置和其他Tween动画很相似，惟一的区别在于From和To是一个Vector3变量，起始点的设置都需要分别填入X、Y、Z的值。当我们成功添加了该组件之后，From和To中会自动读取该物体当前的Transform中Position的值作为默认值。

## 4.4.3 使用位移动画的注意点

使用位移动画的注意点可以参看4.2节讲解透明度动画时的一些注意点，Tween动画的很多注意点几乎都是一样的，比如动画如果选择了Loop和PingPong，那么就不会有结束的那一刻。

除此之外，在制作位移动画时，需要特别注意的是位移的单位问题。因为如果物体属于一个UIRoot，那么它的位置信息是相对位置，单位也是像素。如果物体不属于UIRoot，那么它的位置信息是空间位置，单位是米。

在制作位移动画时，动画在执行时会先瞬间改变到起始点，然后开始插值位移，所以，我们一定要注意起始点的位置是否是想要的位置，一般情况下，需要起始点的位置是物体的当前位置。虽然添加完组件之后它会自动读取当前位置信息作为默认的位置，但是，如果改变了物体的位置信息，TweenPosition组件中的值是不会再次自动地去读取的，所以，需要经常检查From和To的值。

## 4.5 旋转变化动画（旋转动画）

### 4.5.1 旋转动画的介绍和应用

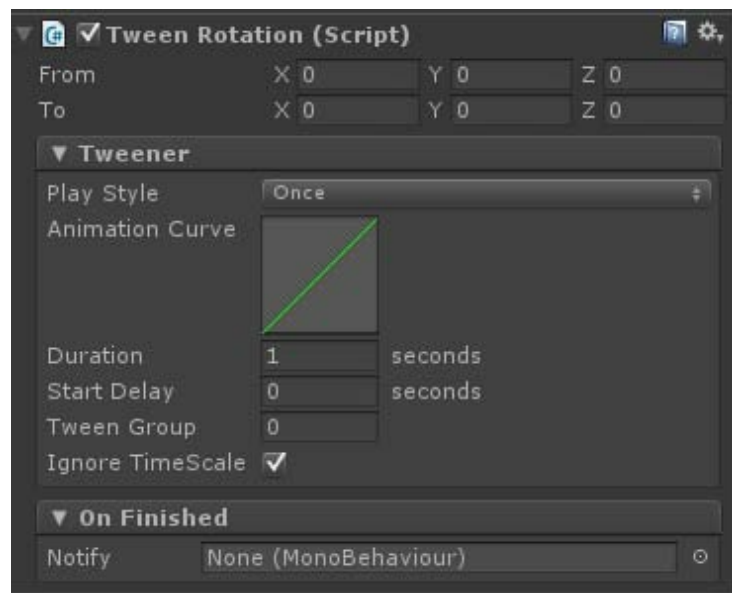
Unity 的每一个 GameObject 都有一个 Transform 组件，每一个 Transform 组件上都有 Position、Rotation、Scale 3 个信息。上文我们讲解的位移动画则是通过改变 Transform 的 Position 来实现的，这里要讲的旋转动画则是通过改变 Transform 组件的 Rotation 来实现的。

旋转动画在游戏中的应用相对要少一些，比如说通过某个操作会使 UI 图片旋转一下。

### 4.5.2 使用旋转动画 TweenRotation

我们需要为某个 UI 增加旋转动画时，可以通过为它增加一个 TweenRotation 组件，这个组件在 NGUI 的 Tween 动画库中，添加方式为，依次选择 AddComponent → NGUI → Tween → TweenRotation。

添加成功后的 TweenRotation 组件设置界面如图 4.11 所示，我们了解一下如何设置它来制作一个旋转动画。



▲ 图4.11

从图 4.11 中可以看到，它的设置界面和位移动画非常相似。相同的一些设置就不多赘述了。主要来讲一下它的 From 和 To 起始点设置。



旋转动画的From和To需要填入的是3个方向的旋转角度，0就是0°，也就是不旋转。需要注意的是，360°，并不等于0度，因为动画有插值算法，例如，在X中填写360，则会顺着X轴的方向将图片翻转360°一整圈。

关于旋转动画的旋转方向，它会根据填入的 X、Y、Z 值合成一个方向（这个 X、Y、Z 值本身就组成了 Vector3 向量），如果希望它是朝着轴正向的，则填入正数，如果希望它是反向的，则填入负数。

例如，设置From的X、Y、Z都为0，也就是以当前原始姿态作为起点，然后在To中的X、Y、Z中分别填写X为60、Y为180、Z为-360。则在持续时间内，它会插值的沿着X轴正向旋转60°，同时朝着Y轴正向旋转180°，同时朝着Z轴的负方向旋转360°，然后在同一个时间点（动画的结束时间）旋转结束。

我们在设置From和To时，虽然填写的是一个旋转度数，我们可以填写超过360的数字，例如，在To中填写720，则是单次动画过程中旋转720°，也就是两圈。

### 4.5.3 使用旋转动画的注意事项

使用旋转动画时，我们要注意以下几点。

（1）动画的方向不要弄错，它是由 X、Y、Z 上的 3 个方向叠加起来的。如果只需要 UI 朝一个方向旋转，请将另外两个方向的值设为 0。

（2）只有From的X、Y、Z都为0时，才是从原始形态开始旋转。

（3）0°和360°区别很大。虽然0°和360°看到的结果是一样的，但是，因为动画有一个插值过渡的过程，所以0°不会旋转，而360°则会旋转一整圈回到原样。

（4）记住正数为正方向，负数为负方向。



(5) 填写的数值可以超过360，例如，填写一个720°旋转，则是单次动画转两圈的意思。

## 4.6 大小变化动画（放缩动画）

### 4.6.1 放缩动画的介绍和应用

放缩动画和旋转动画和位移动画一样，都是通过改变物体的Transform组件来实现的。位移动画是改变的Transform的Position信息，旋转动画改变的是Transform的Rotation信息，而放缩动画，则是改变的Transform的Scale信息。

放缩动画在游戏中比较常用，比如某个图片不停地变大变小造成一种“不停的蹦”的效果。

### 4.6.2 使用放缩动画TweenScale

我们需要为某个UI增加放缩动画时，可以通过为它增加一个TweenScale组件，这个组件在NGUI的Tween动画库中，添加方式为，依次选择 AddCompent → NGUI → Tween → TweenScale。

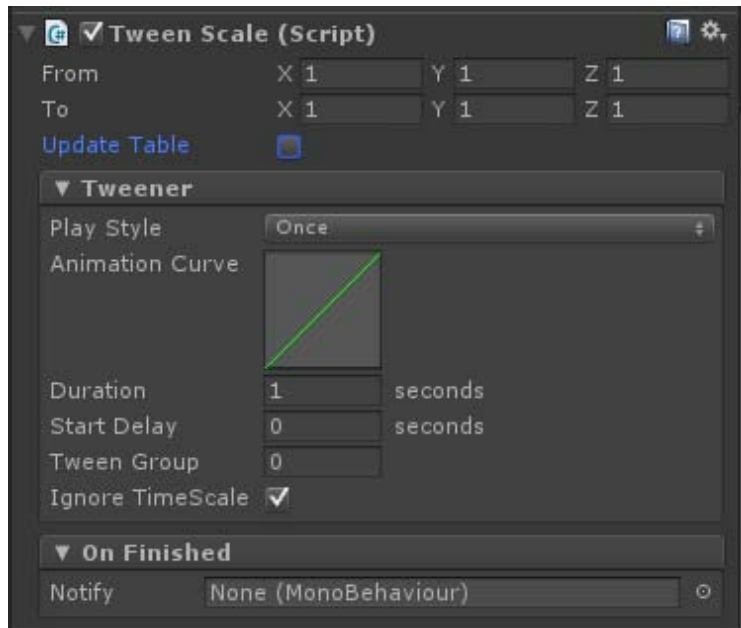
添加成功后的 TweenScale 组件设置界面如图 4.12所示，我们了解一下如何设置它来制作一个放缩动画。

从图4.12中依然看到了一个非常熟悉的设置界面，依然是设置From 和 To 的起始值和Tweener模块，以及OnFinished回调函数模块。这里就不多讲解了。

但是在图4.12中我们发现这个TweenScale组件和其他组件不一样的地方在于多了一个选项：Update Table，这是一个Bool 型变量，打勾为

选中，不打勾则为不选。这个选项在开发时暂时没什么用，可以不用管它。

值得注意的是，TweenScale改变的是LocalScale。



▲ 图4.12

### 4.6.3 使用放缩动画的注意事项

放缩动画比较简单，在使用时注意两点即可：①它改变的是LocalScale。②它会连同子物体一起给改变了。

## 4.7 Tween动画总结

在前几个小节中，我们已经讲解了游戏开发中最常用的一些UI动画效果。在NGUI中还提供了一些其他的Tween动画，比如说：改变音量的动画、整体Transform改变的动画、摄像机的正交尺寸动画等。

因为NGUI的Tween动画的原理和结构几乎都一样，所以，对于其他的Tween动画我们就不多讲述了，大家有兴趣的可以去尝试一下。我

们现在来总结一下Tween动画的特点。

(1) 都要设置From和To，也就是动画的起点和终点，然后动画会根据这两个点，插值出中间过渡点，最后完成起点到终点的过程。

(2) 都有一个Tweener模块，里面需要设置动画的曲线、持续时间等。

(3) 都有一个OnFinished模块，可以设置动画播放完成之后触发的事件函数。

(4) Tween动画都是实实在在地去改变UI的相关参数，并不是一个“临时效果”。

Tween动画一般很少单独使用，一般来说它都需要配合UIPlayTween（一种UI动画的控制器）组件来使用，甚至是和代码结合起来使用。因为在游戏开发中，UI动画的规则会受游戏规则的一些影响而有不同的需求，如果仅是Tween动画单独使用会有很多缺陷，而配合UIPlayTween则会更加强大和方便，在某些更高级的动画控制需求中，还得用代码去控制动画。

只有在它永久存在的时候（如Loop或PingPong）才会单独使用，例如，需要让一行字永远在UI面板中上下漂浮，那么可以设置一个TweenPosition选择PingPong或者Loop模式直接完成制作，这个Tween动画就是单独使用。

在后面我们就将介绍两个很重要的动画的控制组件：UIPlayTween和UIPlayAnimation。

## [4.8 动画控制组件UIPlayTween](#)

### [4.8.1 为什么要用UIPlayTween](#)

首先，我们需要了解，什么是UIPlayTween？UIPlayTween是一个NGUI的组件脚本，它的作用是对物体身上的Tween动画进行自定义的控制，比如我们之前讲解各种Tween动画时，在Tweener设置模块就有一个Group设置，这个Group就是为UIPlayTween准备的变量，可以让UIPlayTween对动画进行整组控制。

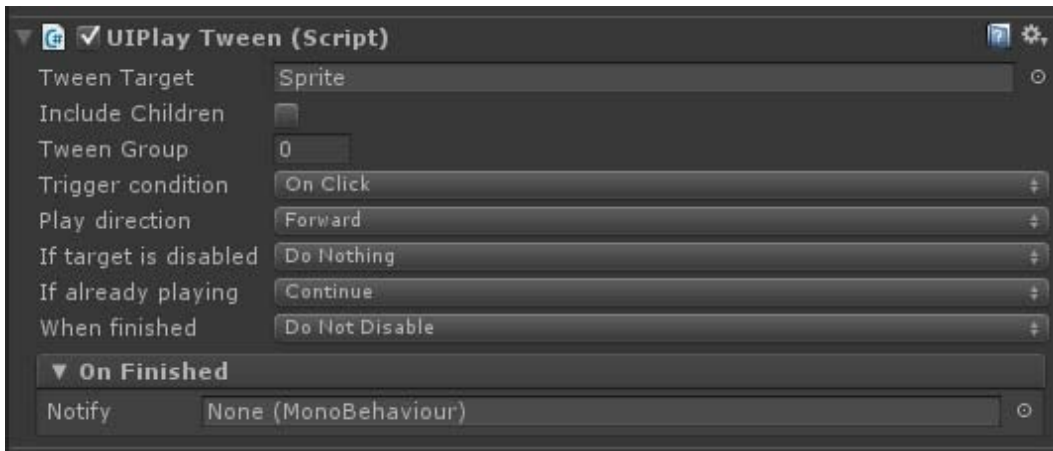
为什么我们要使用UIPlayTween呢？在之前我们讲解各种Tween动画时，讲到了Tween动画的一些特点，这些特点有一部分也是Tween动画单独使用的硬伤，比如Tween动画组件在激活时就会直接播放，播放完一次之后就再也不会播放了。Tween动画如果单独使用，很难满足我们在游戏开发中的很多常见需求，例如，我们需要每次单击某个按钮时，都让一张图片播放一个Tween动画；或者我们需要第一次单击某个按钮，一张图片会正向播放动画，再单击一次按钮，这个图片会倒着播放动画，再单击再正向播放.....这些需求都是Tween动画单独使用无法做到的。

综上，我们需要对动画进行事件关联的时候，也就是我们需要让动画按照一些规则来进行播放的话，我们一般就需要结合UIPlayTween组件来使用。对于某些更高级的需求，甚至需要通过代码来控制。

### [4.8.2 动画核心组件UIPlayTween讲解](#)

UIPlayTween 组件我们可通过依次选择AddCompent → NGUI → Interaction → PlayTween来完成添加。值得注意的是：UIPlayTween 需要接收外部单击事件，所以一般来说，都会将UIPlayTween放在一个有BoxCollider可以接收事件的物体上。

图4.13是UIPlayTween的组件设置界面，先来了解一下UIPlayTween组件的设置方式和特点。



▲ 图4.13

### 1. TweenTarget

动画播放目标物体。

我们刚讲过UIPlayTween 组件的核心意义在于控制物体身上的各个 Tween 动画组件，而这个设置项的意义在于设定 UIPlayTween 组件控制的是哪一个物体身上的 Tween 动画组件。它的默认值是自身的物体，但是，它不一定非得是自身，它可以控制其他的物体。

例如，假设我们希望通过单击按钮A来使物体B播放身上的 Tween 动画，我们需要将B物体拖动到A物体的UIPlayTween组件的 TweenTarget 设置处，这样就将这个 PlayTween 组件和物体B关联起来了。

一个UIPlayTween 只能关联一个物体，关联的物体身上的 Tween 动画组件不管是否激活都可以被 UIPlayTween 控制，一般情况下都会提前关闭物体身上的 Tween 动画组件，让UIPlayTween组件按照事件来控制 Tween 动画的激活与播放。

### 2. Include Children

是否包含子物体。

这里指的是UIPlayTween 组件是否控制目标物体的子物体身上的 Tween 动画。如果不勾选，则UIPlayTween只会控制目标物体身上的

Tween动画播放；如果勾选，则会使UIPlayTween不但控制目标物体身上的Tween动画，而且会同时控制目标物体的子物体身上的Tween动画。

### 3. Tween Group

控制的Tween动画的组。

UIPlayTween组件虽然是控制目标物体身上的Tween动画组件的播放，但是，它并不是直接控制目标物体身上所有的Tween动画。我们在前文讲解Tween动画组件的时候，讲到过每一个Tween动画都有一个Group设置，UIPlayTween组件身上也有一个TweenGroup的设置，它只会控制目标物体身上Group和自身的TweenGroup相同的Tween动画。

例如，目标物体身上有A、B、C3个Tween动画组件，它们的Group分别是0、1、2，而UIPlayTween组件的TweenGroup为0，那么UIPlayTween将只会控制目标物体身上的A动画组件，因为Group一样。

### 4. Trigger condition

触发机制。

在这个设置项中，提供了很多选项，如OnClick、OnPress等，可以用来设定在什么事件条件下使相关联的Tween进行播放。但是，我们之前讲过UIPlayTween所在的物体必须要有BoxCollider组件才能进行事件接收。

### 5. Play direction

播放的方向。这里提供了3种方向。

#### ●Forward

正向播放，默认方向。也就是Tween动画将会正常地从From播放到To。

#### ●Toggle

开关播放，也就是第一次单击，Tween动画会正常地从From播放到To。第二次单击会从To倒着播放到From，再次单击则再从From播放到

To。如此往返循环，像开关一样。

- Reverse

反向播放，Tween动画将会倒着从To播放回From。

## 6. If target is disabled

如果要对目标物体进行播放动画的控制时，目标物体却被禁用了（未激活）状态下的设置。这里提供了以下选项。

- DoNothing

不做任何处理，也就是目标没有被激活，那么该次控制就无效。

- Enable Then Play

强行将目标物体激活，然后播放相应的Tween动画。

需要注意的是，这里是针对的目标物体是否激活，而不是目标物体身上的 Tween 动画组件是否激活。UIPlayTween在工作时，不管目标身上的Tween动画组件是否激活，UIPlayTween组件都会强行控制Tween动画组件激活并按照规定播放动画。

## 7. If already playing

如果对目标物体进行播放动画时，目标动画恰好正在播放中的设置。这里提供了以下选项。

- Continue

接着播放，其实相当于不做任何处理，任由它继续播放完。

- Restart

重新播放，强行将正在播放的动画回到起点，然后按照该次控制重新播放一遍。

- Restart If Not Playing

等这次播放完了，再重新返回到起点播放一遍。

## 8. When finished

播放完毕之后目标物体的处理。这里给出了以下选项。

- Do Not Disable



不要禁用目标物体。其实就是不做任何处理，播放完了就完了。

- Disable After Forward

在正向播放完毕之后，就把目标物体禁用了。

- Disable After Reverse

在反向播放完毕之后，就把目标物体禁用了。

## 9. On Finished

动画播放完毕之后的触发事件函数。设置方法就不多讲了，前面讲解控件时讲过。

需要注意的是，我们一般都会将动画播放完毕之后的回调函数放到这里，而不是放到单个Tween的OnFinished中。

### 4.8.3 使用UIPlayTween的注意事项

UIPlayTween是一个非常有用的组件，它几乎和Tween动画形影不离，它不仅可以整组控制Tween动画的播放，还可以按照事件来触发Tween动画的播放，还有丰富的辅助功能。但是，我们在使用UIPlayTween来进行动画控制的时候，要注意以下一些事情。

(1) 最重要的一条，UIPlayTween需要接收各种事件，所以，确保它所在的物体身上有BoxCollider组件。

(2) 千万要注意UIPlayTween的TweenGroup和目标物体身上Tween动画的Group。因为，它们的默认值都为0，如果不注意的话，它会直接把目标物体身上所有的Tween动画一起控制了。

(3) 谨慎使用Include Children，特别是在还不确定子物体将会有哪些Tween动画时。

(4) 一般来说，假设我们需要一批 Tween 动画都播放结束后，触发某事件函数，最好将函数挂在UIPlayTween的OnFinished中，利于维护。

(5) 如果我们不事先禁用目标物体身上的 **Tween** 动画，那么它在物体激活时就会直接开始播放（也就是**Tween**动画组件自身的规则它会照常执行），所以一般情况下，我们会事先禁用目标物体身上的**Tween**动画组件，让它完全交由**UIPlayTween**来控制激活组件和播放。

## **4.9 动画控制组件UIPlayAnimation**

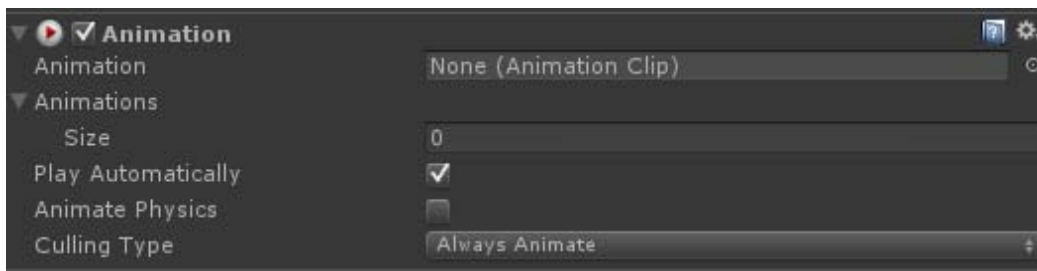
### **4.9.1 为什么要用UIPlayAnimation**

我们还是首先来了解一下什么是**UIPlayAnimation**。**UIPlayAnimation**是NGUI自带的一个动画控制的脚本组件。它和**UIPlayTween**几乎是一模一样的功能，最大的区别是**UIPlayTween**是控制目标物体身上的**Tween**动画，而**UIPlayAnimation**是控制目标物体身上的**Animation**动画。

我们知道**Animation**动画是利用Unity自带的**Animation**系统制作出的**AnimationClip**（动画剪辑），然后将这个**AnimationClip**挂到物体身上的**Animation**组件中（这个组件下面会讲如何添加），实现和**Tween**动画一样的用途。**UIPlayAnimation**的出现就是为了控制目标物体身上的**AnimationClip**。

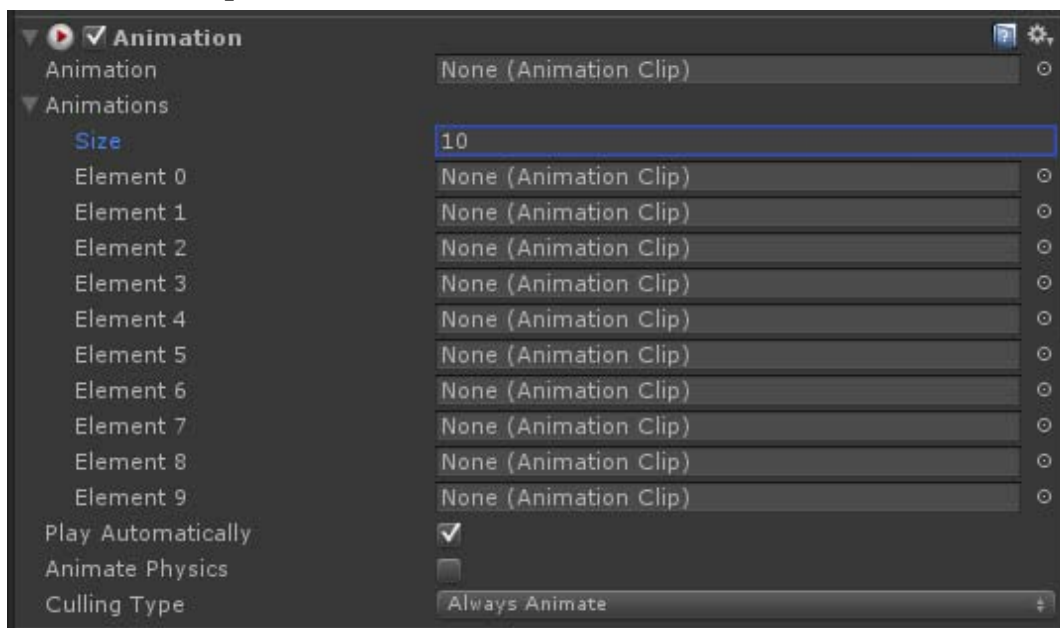
### **4.9.2 为UI添加Animation组件**

为UI添加**Animation**组件的方法是，依次选择 **AddComponent** → **Miscellaneous** → **Animation**，然后得到如图4.14所示的**Animation**组件，将**AnimationClip**文件从**Project**视图中拖动到第一个**AnimationClip**设置项中即可完成默认动画的设置。



▲ 图4.14

需要注意的是，在使用Tween动画时，如果要为UI添加多个Tween动画，我们的方法是添加多个Tween动画组件。而使用AnimationClip时，如果我们要为UI添加多个AnimationClip，则我们只需要使用一个Aniamtion组件即可，只需要将Animation组件中的Size（这个选项代表着该物体身上AnimationClip的个数）设为一个想要的数量，就会出现图4.15所示的界面，会多出相应数量的AnimationClip设置项，然后将AnimationClip都拖上去即可。



▲ 图4.15

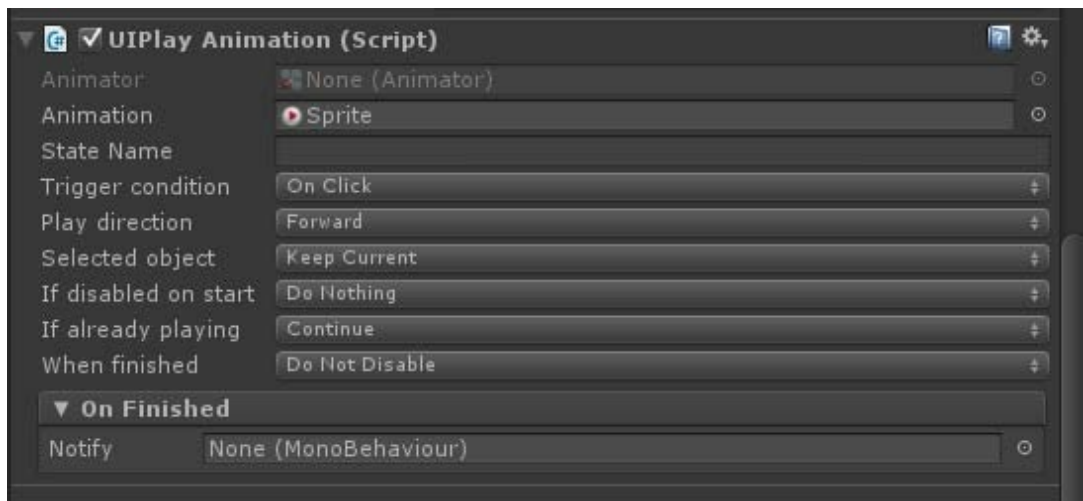
我们最初设置的第一项Animation是默认的Animation，类似于Tween一样，组件激活就会自动播放，就像状态中的默认状态一样，其实这一项Animation不设置也没有关系。而其他这么多的AnimationClip

怎么来控制呢？在旧版本的Unity中，程序员一般是通过代码来控制不同的动画播放，而在NGUI中，因为这些动画都是针对UI的，所以，可以利用NGUI自带的UIPlayAnimation更方便地进行控制。

### 4.9.3 动画核心组件UIPlayAnimation讲解

再次强调，虽然 AnimationClip 是通过 Unity 的 Animation 系统制作出来的，但是UIPlayAnimation组件是NGUI的一个组件。它的添加方法为：依次选择AddCompent → NGUI → Interaction → PlayAnimation。

UIPlayAnimation组件的设置界面如图4.16所示，我们先来了解一下这个组件，虽然它的功能和UIPlayTween大体差不多，但是在设置上还是有一些区别。



▲ 图4.16

#### 1. Animation

第一行的 Animator 我们不需要设置也无法设置，这里就不讲了。需要设置的第一个项是Animation，这里设置的其实是一个带有Animation组件的物体，并不是一个AnimationClip！

我们需要确保这个UIPlayAnimation组件控制的目标物体身上有一个Animation组件（上一小节讲到的组件），然后将这个目标物体拖动

到这个Animation设置项中来，即可完成设置，和UIPlayTween中设置Target是一样的原理。

## 2. StateName

需要播放的动画片段的名称。

我们也许在目标物体身上的 Animation 组件中添加了很多的 AnimationClip，而这里的StateName就是填写要播放哪一个动画，填入的是动画的名称，必须保证这里填入的名称和目标物体身上相应的 AnimationClip的名称完全一致，包括大小写。

## 3. Trigger condition

和UIPlayTween一样。

## 4. Play direction

和UIPlayTween一样。

## 5. Selected object

对选中的物体的设置，一般保持它为默认的KeepCurrent即可。

## 6. If disabled on start

和UIPlayTween一样。

## 7. If already playing

和UIPlayTween一样。

## 8. When finished

和UIPlayTween一样。

## 9. Onfinished

和UIPlayTween一样。

### 4.9.4 使用UIPlayAnimation注意事项

UIPlayAnimation在使用的时候，和UIPlayTween还是有一些区别，我们在使用时，要注意以下一些事项。

(1) 最重要的一条，UIPlayAnimation 需要接收各种事件，所以，确保它所在的物体身上有BoxCollider组件。

(2) UIPlayAnimation没有Group，目标物体的Animation组件中也没有Group，所以，单个UIPlayAnimation组件是无法整组控制一批AnimationClip的播放。

(3) 确保目标物体身上有 Animation 组件，并且确保UIPlayAnimation 组件中填写的StateName和目标物体身上Animation组件中的相应的AnimationClip名称完全一致，因为，它的判断方式是字符串相等。

(4) 使用 Animation 来制作 UI 的动画时，我们一般不需要事先就禁用目标物体身上的Animation组件。

# 第5章 其他组件

## 5.1 使用Toggle制作页签

### 5.1.1 页签的工作原理

页签的功能，相信大家都非常常见，页签会将内容分类来显示，同一时间内，只能选中一个页签，并且只显示该页签所属的内容。这个功能和我们讲解Toggle制作复选框时有些相似，其实页签就是一个单选框，由多个同组的开关（Toggle）组成，同一时间只能选中一个选项。

页签和这个单选框功能的区别就是，页签中包含有要显示的内容。当切换开关时，不但要自动将其他开关关闭，并且还要将当前选中的开关所包含的内容显示出来，这就是页签的功能原理。

在制作页签时，我们的思路也是先使用Toggle制作一系列同组的开关（单选框），然后为每一个开关再增加一个ToggleObjects组件，通过ToggleObjects组件来设定每个开关包含的内容。当该开关没有被激活时（即该页签没有被选中），它所包含的内容会自动隐藏（包含的物体被Disable掉），当该开关被选中时，它所包含的内容会自动激活显示。

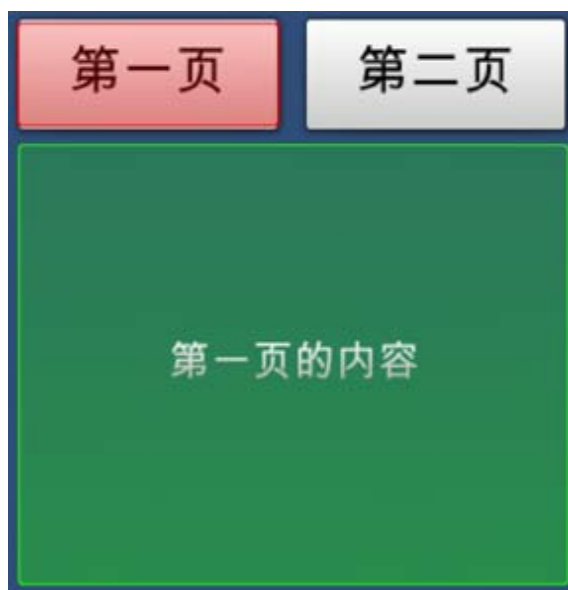
### 5.1.2 一个完整的页签界面



下面我们就来试着做一个简单的页签，我们将要做的页签效果如图5.1和图5.2所示。

### 5.1.3 制作两个页签按钮

为了制作出如图5.1和图5.2所示的页签界面，我们从图中观察得知，这个范例界面一共只有两页，所以先来制作两个页签按钮。



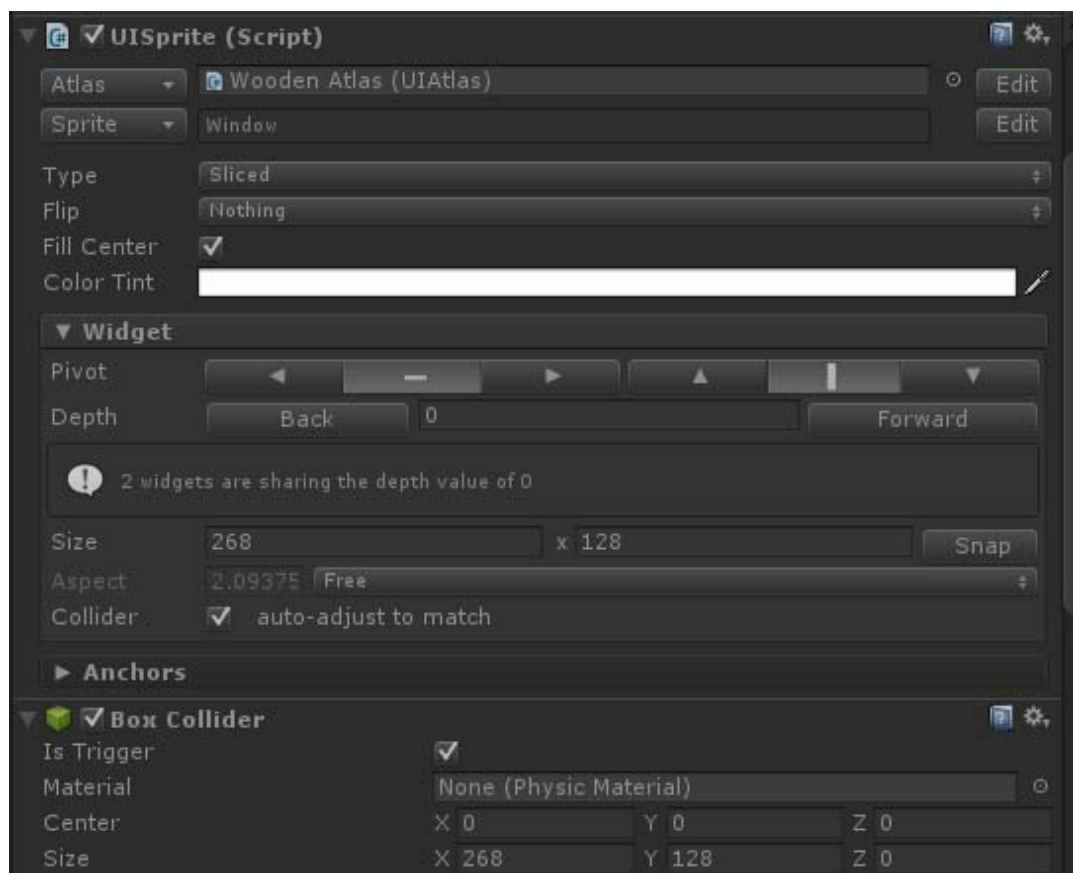
▲ 图5.1



▲ 图5.2

(1) 先创建一个3DUI（2DUI也可以），然后在UIRoot下生成两个Sprite，分别将它们命名为Button1和Button2。

(2) 选择NGUI自带的WoodenAtlas中的window作为Sprite的图片资源，然后将Sprite改为 Sliced 模式，设定它的尺寸到想要的大小。然后为 Button1 和 Button2 分别 Attach 一个BoxCollider。Inspector面板如图5.3所示。

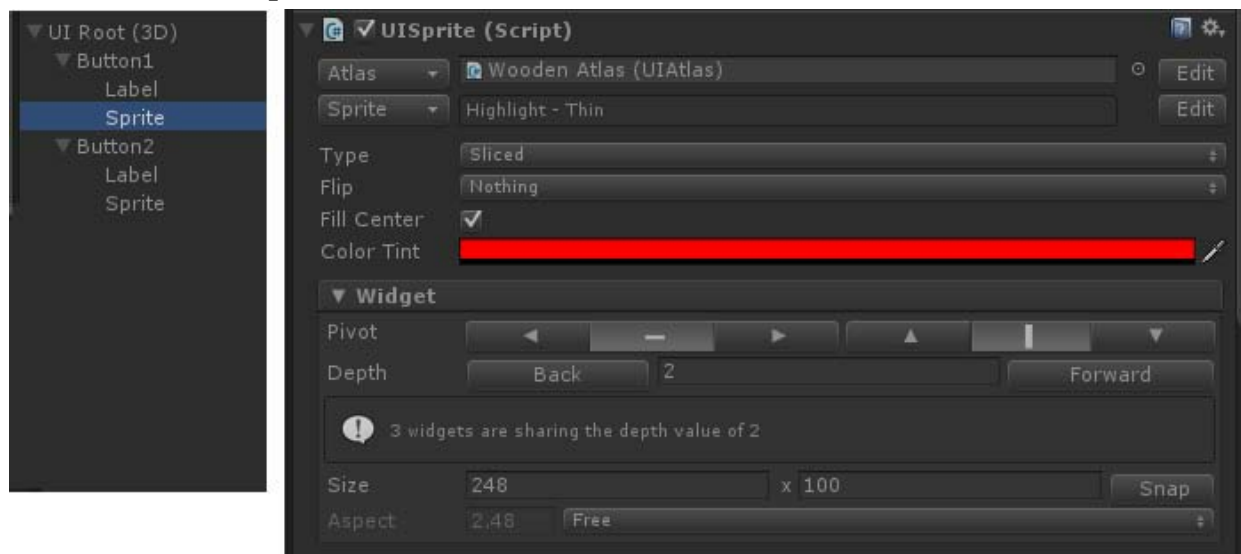


▲ 图5.3

(3) 分别在Button1和Button2下创建一个Label作为子物体，Button1下的Label的文本写上“第一页”，Button2下的Label的文本写上“第二页”。

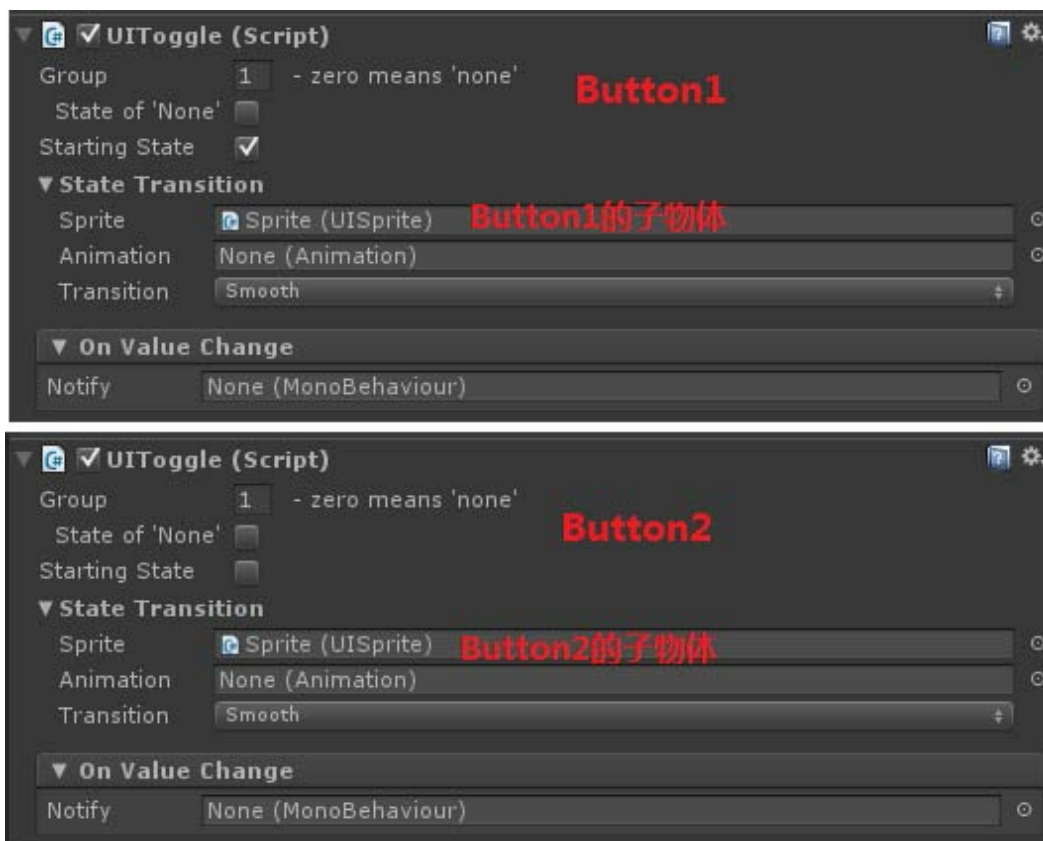
(4) 分别在Button1和Button2下创建一个Sprite作为子物体，这个子物体将来表示该页签被选中了，我们选用NGUI自带的WoodenAtlas

中的Highlight-Thin作为Sprite的图片，并改变它的颜色为纯红色。UI结构和子物体的Sprite组件如图5.4所示。



▲ 图5.4

(5) 分别为Button1和Button2附加一个Toggle组件，将它们的Group都设为1，并且将Button1 的Toggle 的Starting State 勾上，表示Button1 为默认显示的页签。然后将Button1 和Button2下面用来表示选中状态的Sprite子物体分别拖到Button1和Button2的Toggle组件的Sprite设置项中，这里注意不要弄混了。Button1和Button2的Toggle组件如图5.5所示。



▲ 图5.5

到此为止，就制作好了两个页签按钮，我们可以看到它们看上去都处于激活状态，不过没有关系，运行游戏时，它们就会变成正常状态，可以自由地切换，如图5.6所示。



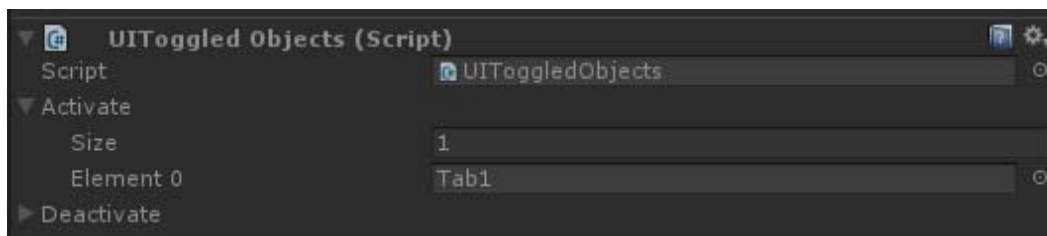
▲ 图5.6

### 5.1.4 使用ToggleObjects来记录页签内容

我们首先需要制作两个页签所需要显示的页面，从图5.1和图5.2可以看到，两个页签需要显示的内容很简单，页签1显示的是一个绿色的面板，上写着“第一页的内容”；页签2显示的是一个红色的面板，上写着“第二页的内容”。

我们先制作两个Sprite分别命名为Tab1和Tab2，然后使用NGUI自带的WoodenAtlas的Highlight-Thin作为图片，将它的尺寸设定为需要的大小，调整好颜色，并且分别在这两个Sprite下创建一个Label子物体，文本内容分别写上“第一页的内容”和“第二页的内容”。上一小节我们做好了两个页签按钮，Button1 页签需要显示的内容就是 Tab1，Button2 需要显示的内容为Tab2。

然后为Button1和Button2这两个页签按钮分别增加一个组件：ToggleObjects。增加方法为：依次选择AddComponent → NGUI → Interaction → ToggleObjects。组件界面如图5.7所示。



▲ 图5.7

在Activate设置中，就可以设置这个开关被选中时，需要显示的内容，这些内容在该页签没有被选中时会自动隐藏，只有当该页签被选中时，才会自动显示出来。

在Activate设置中，可以设定该页签包含的内容有多少个物体，目前情况下，每个页签只需要显示一个物体即可，所以将Size设为1，然后会多出一个Element0，让我们设定第一个内容元素。我们将Tab1物体

拖动到Button1的UIToggledObjects的Element0中，同理将Tab2物体拖动到Button2的UIToggledObjects的Element0中，就算完成了页签包含内容的设定。

我们的页签到此就算制作完成，制作完后，从Game视图中看到的情况如图5.8所示，可以看到页签的内容都重叠在一起了，没有关系，在运行游戏时，没有被选中的页签会自动隐藏掉自己需要显示的内容。



▲ 图5.8

### 5.1.5 制作页签注意事项

制作页签时，我们需要注意以下事项。

- (1) 确保每个页签身上都有BoxCollider。
- (2) 确保每个页签身上都有UIToggle和UIToggledObjects两个脚本。
- (3) 确保所有的页签身上的Toggle的Group一致，且不为0。

(4) 我们并不需要提前去禁用某些物体或组件，一切都会在运行游戏时自动处理。

(5) 这并不是制作页签功能的惟一准则，当我们明白了页签的原理后，可以自由按照自己的想法去实现页签效果。

## 5.2 拖动摄像机来浏览超大界面

### 5.2.1 拖动相机功能的介绍和应用

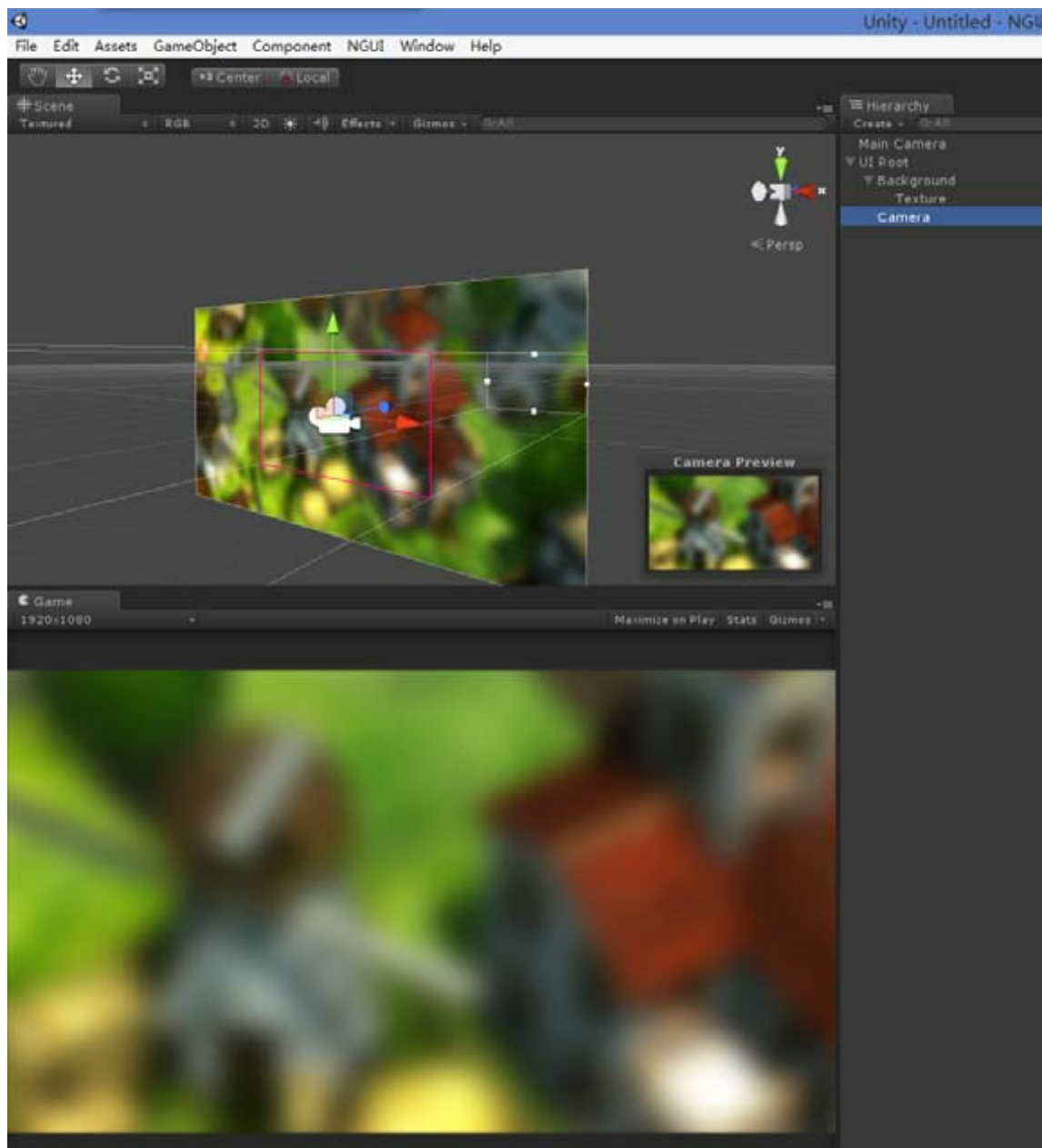
所谓超大界面，一般来说，指一个完整的界面非常庞大，视图大小已经超过了摄像机的照射范围，也就是一屏幕之内无法完全显示，一般来说这种超大界面应用于2D游戏中，经常作为一个Scene的背景图存在。

超大界面在游戏中有着广泛的运用，例如，世界地图的查看，因为世界地图很大，需要拖动地图来进行查看。再比如SLG（经营策略类）游戏中，经常会涉及主城场景视图的拖动。

我们之前讲解ScrollView时，讲过ScrollView的主要目的是为了让一个Panel显示不完的内容可以通过滑动来方便地进行浏览。但是，这里的情况不是特别一样，本节讲解的超大界面，一般是整个屏幕都用来显示这个界面（这个界面一般都是背景），而且还显示不完全，我们浏览时，需要借助拖动摄像机来自由浏览，而且摄像机还不能被拖出边界（否则会看到界面以外的东西）。

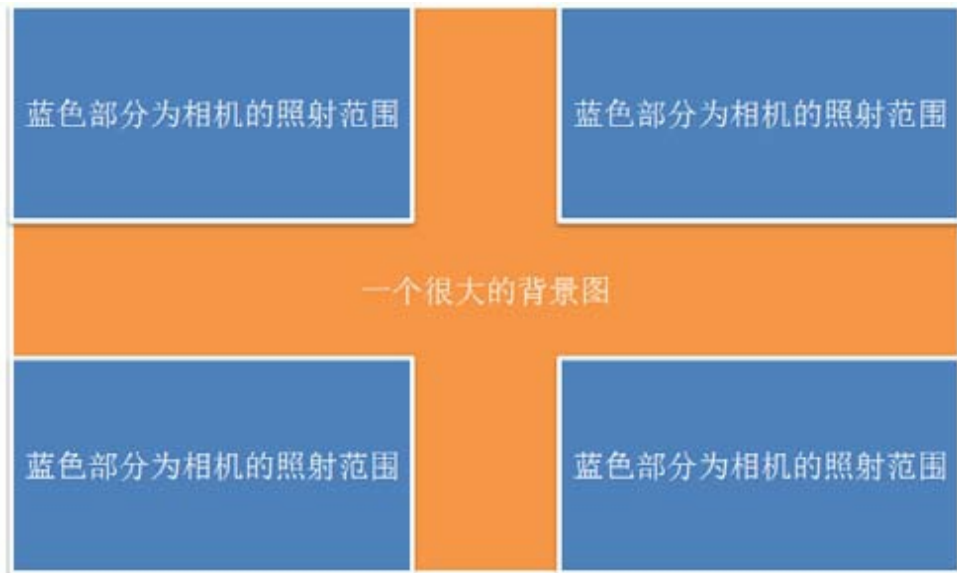
图5.9展示了本章需要讲解的超大界面浏览的一个实例：图5.9中的背景图尺寸大小远远超过了2D正交相机的照射范围，需要快速实现拖动摄像机来浏览整个背景图的功能，同时要求摄像机不能照射到背景图之外的地方，否则会画面穿帮。





▲ 图5.9

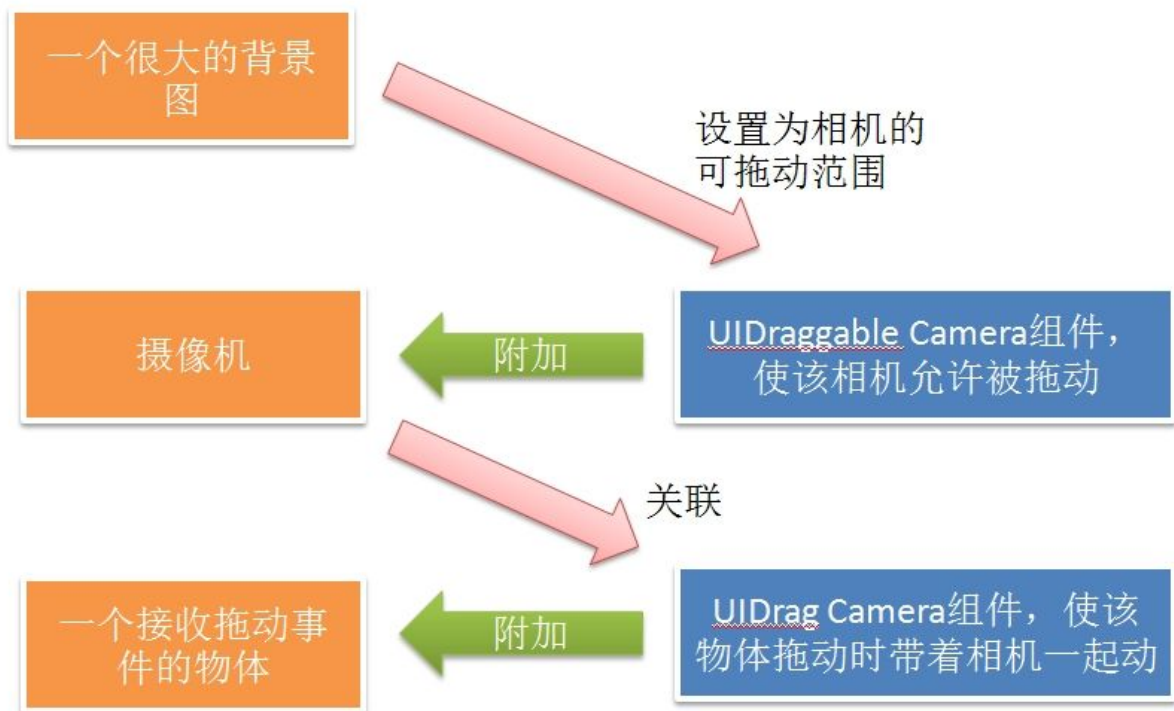
图5.10展示了本节要讲解的功能中，相机的拖动边界的概念。  
图5.10中4个蓝色部分，就是该功能下相机的4个极限位置。



▲ 图5.10

### 5.2.2 核心原理和组件介绍

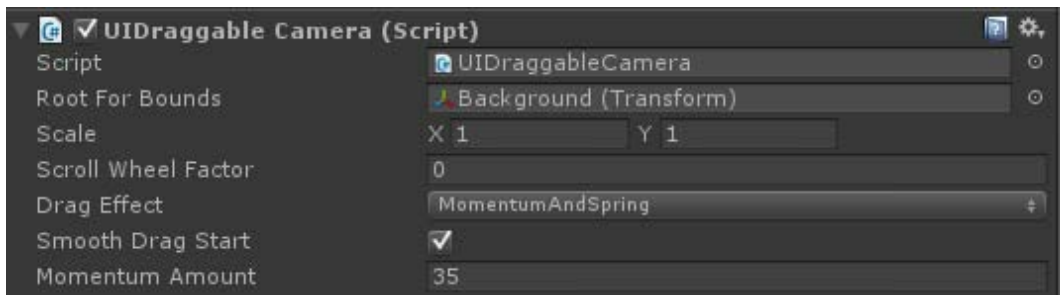
核心组件原理如图5.11所示。



▲ 图5.11

如图5.11所示，可以看到该功能主要用到了两个核心组件：UIDraggable Camera 和UIDrag Camera。

首先来了解一下 UIDraggable Camera 组件，该组件必须添加到摄像机物体上，它的作用是使附着的摄像机变为可拖动的。该组件的添加方式为：依次选择AddCompent → NGUI → Interaction → Draggable Camera。UIDraggable Camera 组件的设置界面如图5.12 所示，我们来了解一下组件的设置。



▲ 图5.12

### 1. Root For Bounds

这是最重要的一个设置项：相机拖动的范围边界。

在设置这个选项时，我们不能直接将背景图拖进去，需要将背景图的根物体拖进去（可以是父物体），如图5.9中，我们的背景图是 Texture，但是，我们设置摄像机的拖动边界时，会把 Texture 的父物体 Background 拖进去，这样它将会自动计算出这个物体包含的内容的边界（NGUITools类中有类似函数，一共会得出minX、maxX、minY、maxY4个值），来限定摄像机的移动边界。

### 2. Scale

可以理解为相机被拖动时在X和Y上的速度。

### 3. ScrollWheelFactor

讲解ScrollView时讲过，鼠标滚轮的影响系数。

### 4. Drag Effect

和ScrollView一样，详情参看ScrollView的讲解。

为了保证相机完全不被拖到边界以外，我们一般选择 **None**，也就是没有拖动效果。否则惯性和弹性可能会导致相机拖动时看到背景图以外的内容。

### 5. Smooth Drag Start

和**ScrollView**一样，详情参看**ScrollView**的讲解。

### 6. Momentum Amount

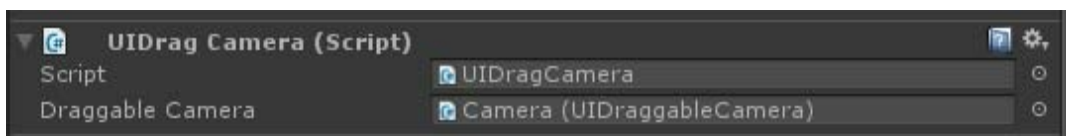
和**ScrollView**一样，详情参看**ScrollView**的讲解。

然后我们再来了解一下**Drag Camera** 组件，该组件一般附着在一个具有**BoxCollider**的物体上，以此来接收拖动事件，来使摄像机拖动，但是一定要注意：这个物体本身是不会被拖动的，因为我们无法看到相机也无法直接操作相机，所以我们需要这么一个超过相机视图面积的**BoxCollider**来接收到的拖动事件，它的拖动事件会被

**DraggableCamera**和**DragCamera**两个组件转化为了相机的拖动事件。

通常情况下，我们可以将该组件附着在背景图上，并为背景图附加 **BoxCollider**，让背景图来充当拖动摄像机的事件接收者，因为背景图足够大并永远不会消失在视野中（相机会被设定边界，不能拖到背景图边界以外的地方）。不过也可以新创建一个空的 **GameObject**，通过**AddComponent** → **Physics** → **BoxCollider**为它附上一个很大（比相机的移动范围还大，永远在相机的照射范围内）的**BoxCollider**，然后为它附上**DragCamera**组件，以这个空物体来作为相机的拖动事件的接收体。

本文以背景图作为相机拖动事件的接收体来作为案例讲解。我们为背景图 **Attach** 一个**BoxCollider**，然后为它附加一个**DragCamera**组件，方法为：依次选择**AddComponent** → **NGUI** → **Interaction** → **Drag Camera**。这个组件的设置界面如图5.13 所示，我们先来了解一下它的设置。



▲ 图5.13

从图5.13中可以看到，Drag Camera 组件的设置非常简单，将我们要拖动的摄像机拖动到DraggableCamera 设置项中即可，但是注意，必须确保被设置的摄像机身上有DraggableCamera组件。

然后运行游戏，我们拖动背景图，可以看到相机也被拖动了，并且相机永远都在背景图范围内照射，这样就可以拖动来浏览整个背景视图的全貌了。

需要注意的是，虽然我们是拖动背景图来实现相机的拖动，但其实背景图本质上是没动的！我们对背景图的拖动事件，被UIDraggableCamera和UIDragCamera两个组件转化为了相机的拖动事件。

### [5.2.3 拖动相机浏览超大界面的注意事项](#)

这个功能在制作2D游戏中有非常广泛的应用。我们在使用这个功能时，需要注意以下事项。

- (1) 尽量确保相机是正交相机，这样可以完美地得到图片边界。一定要保证相机的窗口大小没有超出背景大小。
- (2) 在整个 Scene大背景都需要被浏览时，才使用这个功能。界面内部的拖动浏览还是使用ScrollView更好一些。
- (3) 一定要设置背景物体的父物体作为DraggableCamera组件的边界设置，这样可以避免很多问题。
- (4) DraggableCamera的拖动效果最好设为None，否则拖动到边界时，摄像机会因为惯性超出边界。

(5) 确保各组件正常关联，接收拖曳事件的背景图（或一个比背景图更大的空物体）有BoxCollider和Drag Camera，并设置了具体要拖动哪一个摄像机。

(6) 深刻理解这个方法和ScrollView的原理上的不一样：ScrollView是拖动剪辑视窗内的内容来完成拖动浏览。而本文中讲解的方法虽然拖动的是要浏览的内容，但是，拖动事件被转化为了相机的拖动事件，所以，内容其实没有动，动的是摄像机。

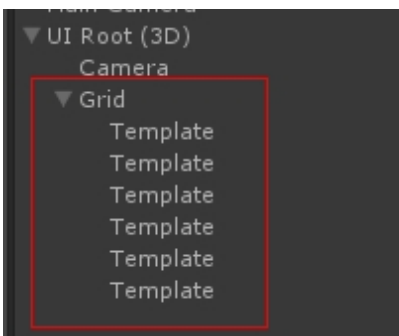
## 5.3 使用Grid自动排列UI

### 5.3.1 自动排列UI的应用

在游戏中，我们会经常碰到UI需要排列的情况，比如我们有4个选项，我们希望它们一字排开，一般情况下我们的做法是手动去定义这4个选项的位置，但是，手动定义的位置的同时，要使每个UI元素之间的间隔相同会比较麻烦，而且如果要排列动态加载的元素那更是比较麻烦。这个时候，就需要一个工具来自动按照一定的间距方式排列UI。

### 5.3.2 自动排列UI核心组件Grid介绍

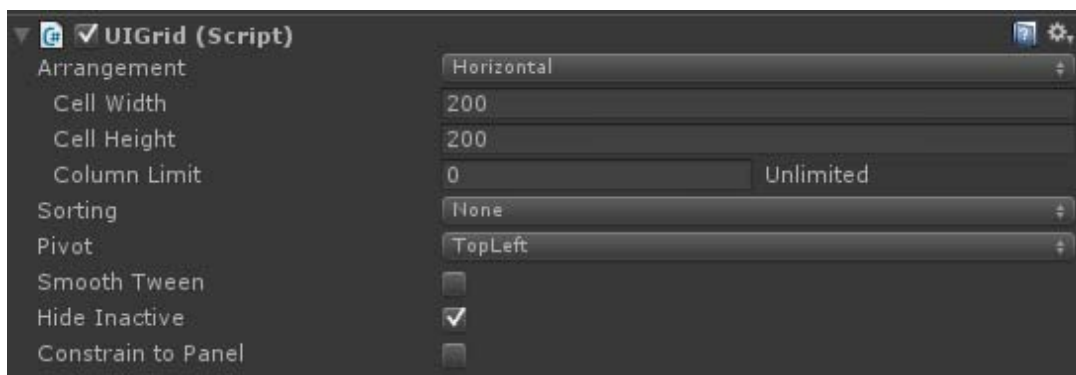
Grid是我们自动排列UI元素的一个常用组件。它的用途是让UI元素按照一定的间距来网格化排列。一般情况下，我们会创建一个Grid物体，然后将需要排列的UI元素放入这个Grid物体下作为子物体存在，如图5.14所示。



▲ 图5.14

Grid的创建方法有两个：第一种方法是选中要创建Grid的UI节点，然后通过Unity顶部NGUI菜单中Creat → Grid即可。第二种方式是创建一个空物体到 UI 节点下，调整它的层和 UI一样，然后为这个空物体增加一个 UIGrid 的组件，添加这个组件的方法为依次选择 AddCompent → NGUI → Interaction → Grid 。

创建好后的Grid组件设置界面如图5.15所示，我们先来了解一下Grid组件的设置。



▲ 图5.15

### 1. Arrangement

这是设置网格的排列方向，目前只支持两种方式：水平排列和纵向排列。

### 2. CellWidth

每一个网格的宽度。



**Grid**的排列原理是先制造出一系列整齐的网格，然后将每个UI元素置于网格中。

### 3. CellHeight

每一个网格的高度

### 4. ColumnLimit

数量限制，默认为 0 无限制。一般情况下，不需要去限制它排列的数量，保持默认就可以。

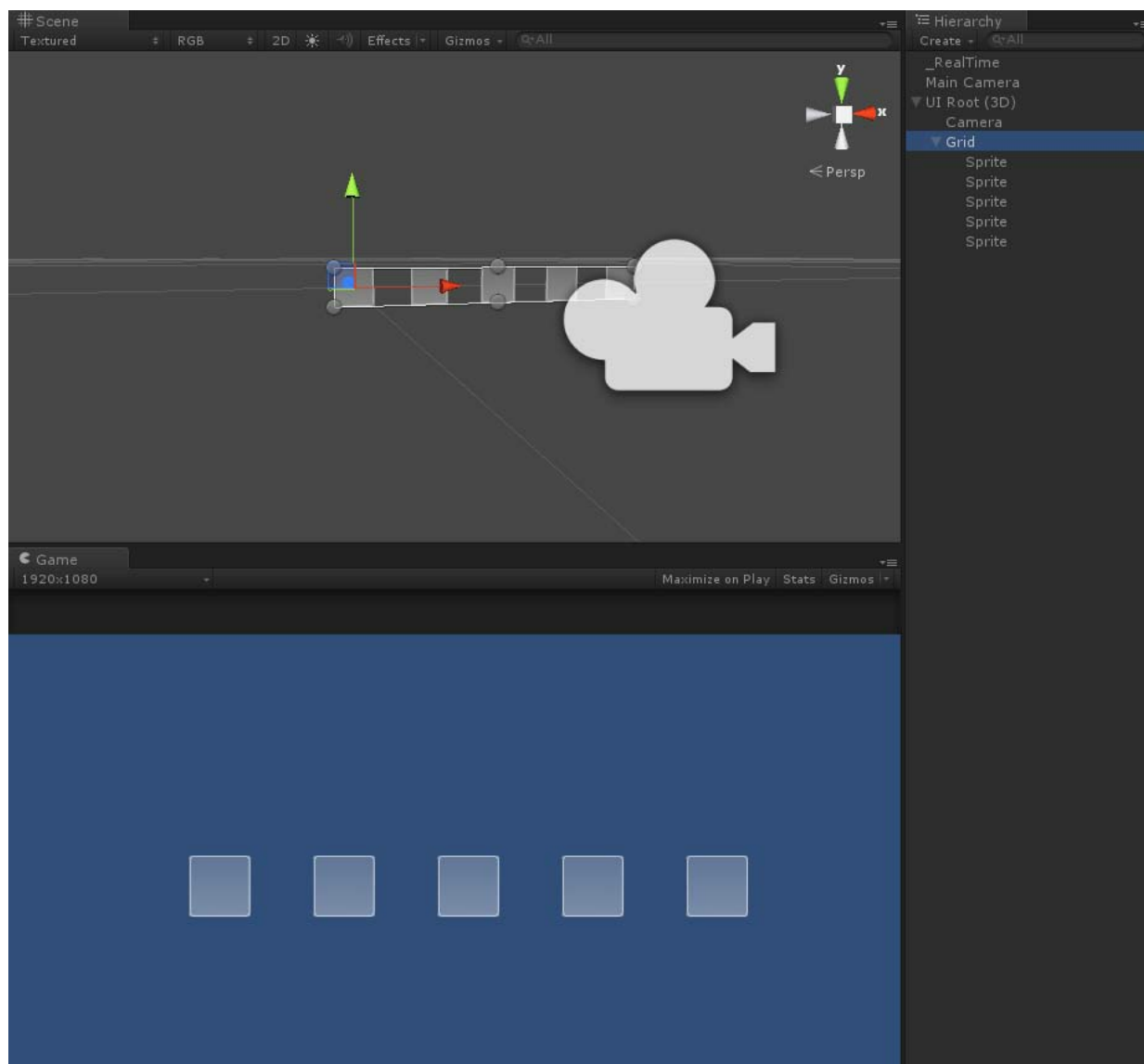
### 5. Pivot

锚点，也就是网格的起始点，一共有9个点。默认为左上角开始排列网格。

### 6. 其他组件

我们一般不需要进行设置。

如图5.16所示，创建了5个Sprite（每个Sprite为一个方块）在Grid下，然后设置Grid为水平排列，每个网格大小为200\*200，图5.16即为运行游戏之后看到的情况。

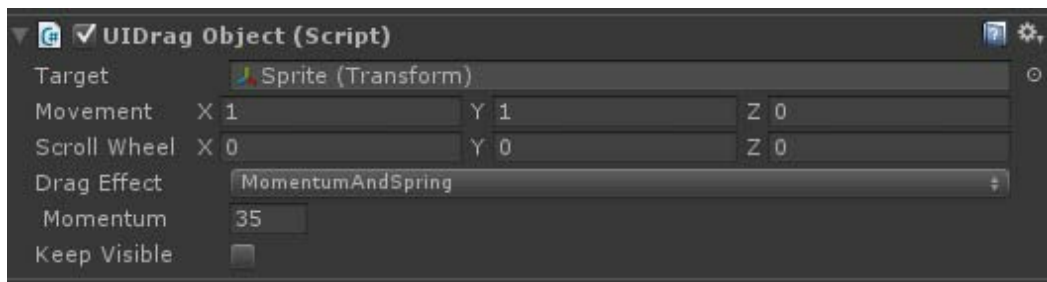


▲ 图5.15

## 5.4 使用DragObject直接拖动物体

当我们需要让某个物体可以任意被拖动时，一般情况下会去写脚本监听用户的拖动操作，然后改变物体的位置。在NGUI中，如果需要拖动某个UI，完全不需要自己写脚本，只需要为该UI物体附上一个接收事件的BoxCollider，然后为它添加一个DragObject组件即可。

DragObject 组件添加的方法为，依次选择 AddComponent → NGUI → Interaction → Drag Object。该组件的设置面板如图5.17所示，我们先来了解一下它的设置。



▲ 图5.17

### 1. Target

目标物体。指的是拖动这个物体，会导致哪一个物体移动。一般情况下，会把物体自身拖动到这里完成设置。

### 2. Movement

移动的灵敏度，可以设置XYZ 的值，一般拖动 UI都是在一个平面内拖动，所以，大部分情况下设置好X和Y的值即可。

### 3. ScrollWheel

滚轮的灵敏系数，需要的话可以设置。

### 4. DragEffect

和ScrollView的拖动效果一样，这里就不多介绍了。

### 5. Momentum

动能，和ScrollView的动能设置一样。

### 6. KeepVisible

让被拖动的UI物体永远在一个矩形内保持可见，一般情况下不需要设置，如果勾选，则会自动多出一个UIRect的设置框。

我们新创建一个Sprite，为它添加BoxCollider和Drag Object组件并完成设置之后，运行游戏，就可以随意拖动它了。

## 5.5 让玩家通过拖动自由改变控件大小

在游戏中，也许会碰到需要让玩家自由拖大、拖小 UI 控件的需求，就像 Windows 系统的各个窗口，我们可以拖动窗口的4个角来自由地改变窗口的尺寸一样。面对这样的需求，一般情况下需要写很复杂的代码，但是在NGUI中，可以很简单地通过添加一个组件实现。

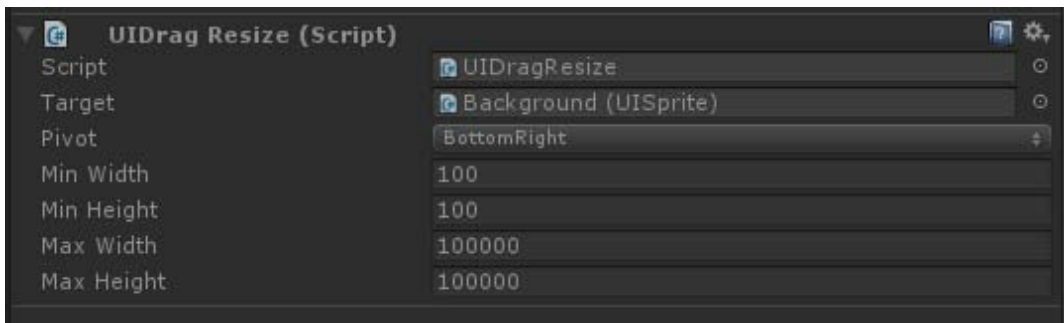
NGUI中提供了一个**DragResize**的组件，将它附着到一个拥有**BoxCollider**的物体上，就可以通过拖动这个物体来改变控件的尺寸。**DragResize** 的组件设置界面如图 5.18 所示，我们先来了解一下组件的设置。

### 1. Target

这是设置拖动这个物体需要改变哪一个控件物体的尺寸？只需要将目标物体拖动到此处即可。

### 2. Pivot

这是设置改变尺寸的锚点，即拖动此物体时，目标物体的尺寸改变是以哪个点开始改变。例如，我们设置锚点为**BottomRight**（底部右方），那么拖动物体时，目标控件的尺寸会由右下点开始改变。

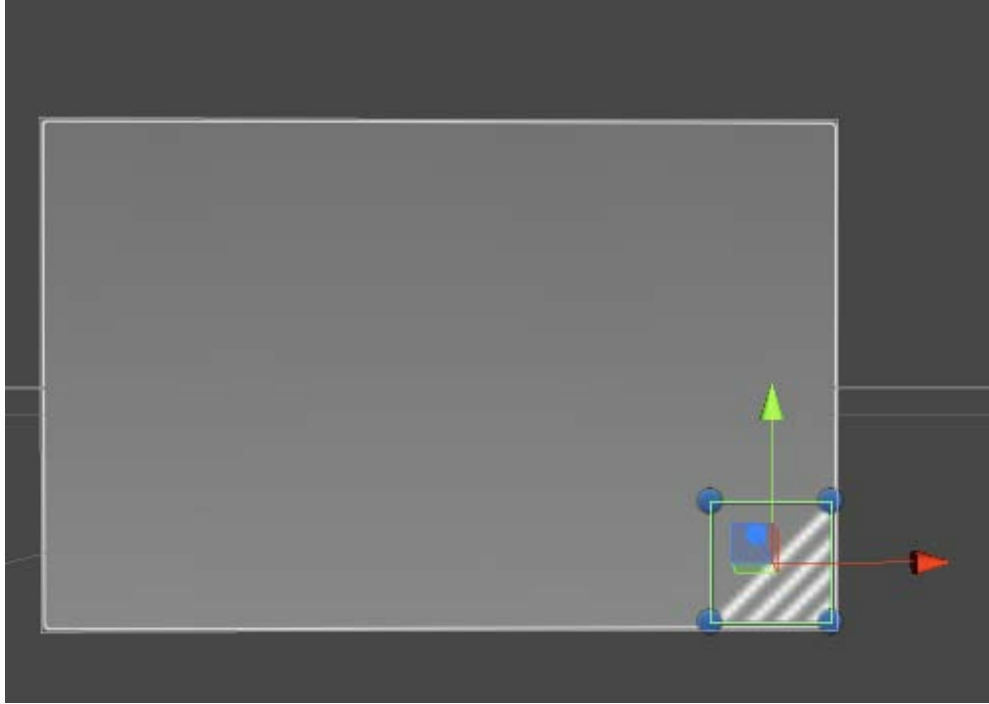


▲ 图5.18

### 3. Min/Max Width, Min/Max Height

设置尺寸改变的最大、最小的宽度和高度，这是为了设置改变控件尺寸的极限值。

我们可以创建一个背景图，在它的右下角创建一个小Sprite作为拖动块，按照上文描述的方法添加了BoxCollider和DragResize，并完成设置之后如图5.19所示，运行游戏，拖动右下角的小Sprite，就可以看到背景图的尺寸被改变了。



▲ 图5.19

但是我们在拖动小Sprite的时候可以看到，小Sprite和背景图的相对位置也变了，这个没有关系，我们在后面讲解UI位置的适配时会详细讲解怎样相对屏幕和物体来固定UI的位置。

## 5.6 制作序列帧精灵动画 (SpriteAnimation)

### 5.6.1 什么是序列帧精灵动画

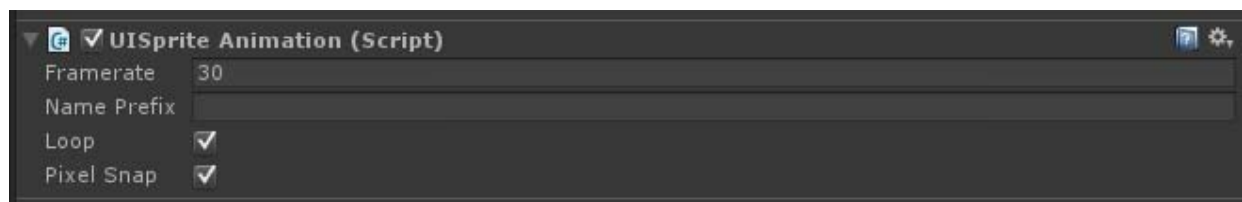
所谓的序列帧动画，就是将动画制作成一系列的单帧图片，然后迅速地依次序替换这些图片达到播放的效果。所谓的序列帧精灵动画，是指将一系列序列帧当成Sprite放置到一个图集内，通过让这个Sprite依次替换来达到播放的效果。

在2D游戏中，所有的动画效果都是序列帧动画，包括了角色动画、特效等。而在3D游戏中，在很多UI特效上，也有可能用到精灵动画。

在NGUI中，提供了一种非常便捷的制作精灵动画的方式，我们可以为任何一个精灵赋予SpriteAnimation组件，即可让这个精灵不停地在图集内按次序替换图片，达到动画效果。

### 5.6.2 SpriteAnimation组件

制作精灵动画，必须确保这个物体上有Sprite组件，也就是说必须在Sprite控件上制作精灵动画。我们将会用到一个NGUI的组件SpriteAnimation，添加方式为：依次选择AddComponent → NGUI → UI → SpriteAnimation。组件界面如图5.20所示。



▲ 图5.20

#### ●Framerate

帧率设置，每秒播放多少帧。这里的帧率和游戏的FPS渲染帧率不是一个概念，这里的帧率是这一套序列帧精灵动画每秒播放多少张。

#### ●NamePrefix

名称前缀，可以限定只播放该**Sprite**图集中带有该前缀名称的精灵。例如，我们将待机动画和跑步动画都放到一个图集里，待机的图片名称前缀全部都是**idle**，跑步动画的图片名称前缀全部都是**run**，这样就可以通过名称前缀的设定让动画只播放图集中的一部分精灵。

- Loop**

是否循环。

- PixelSnap**

是否保持原像素大小进行播放，如果勾选则图集中用到的精灵图片都会在播放时以原像素大小来展现。

使用精灵动画需要了解的还有一旦禁用精灵动画组件，则动画就停了，类似于暂停的效果，如果我们启用这个组件，则动画会接着从之前地方继续播放。除此之外，一定要记住**SpriteAnimation** 组件完全依赖于 **Sprite** 组件，只有在带有 **UISprite** 组件的物体上添加 **SpriteAnimation**组件才有效果。



# 第6章 NGUI实战进阶

## 6.1 UI开发核心问题——UI随屏幕自适应

### 6.1.1 屏幕分辨率对UI适配的影响

我们使用Unity开发的是游戏项目的客户端部分，在开发时就不得不面对一个严峻的问题：不同的显示设备有不同的分辨率和宽高比。这将会对UI的显示造成巨大的影响，例如，我们在讲解UIRoot的时候，讲解过UIRoot的几种缩放UI的方式，如果是不进行缩放完全保持原像素大小，那么在分辨率高的显示屏幕上UI将会显得较小，同理在分辨率低的显示屏幕上UI会显得相对较大。

一般来说，我们UIRoot都会选择FixSize的缩放模式，这样可以让UI随着分辨率而自动缩放，保持和屏幕相对的大小比例不变，让UI整体看上去不会有变大变小的奇怪现象。但是，还有另一个真正严重的问题：不同屏幕的宽高比不一样。

在Unity中，不同屏幕的宽高比，一般都会以高度为基准而拉伸宽度，比如4:3的屏幕换到 16:9 的屏幕上，我们会发现屏幕被拉宽了。如图 6.1 和图 6.2 所示，图 6.1中我们在4:3的屏幕模式下的4个角放上了一个UI图片，然后将屏幕模式切换到16:9下，会出现图6.2所示的画面。

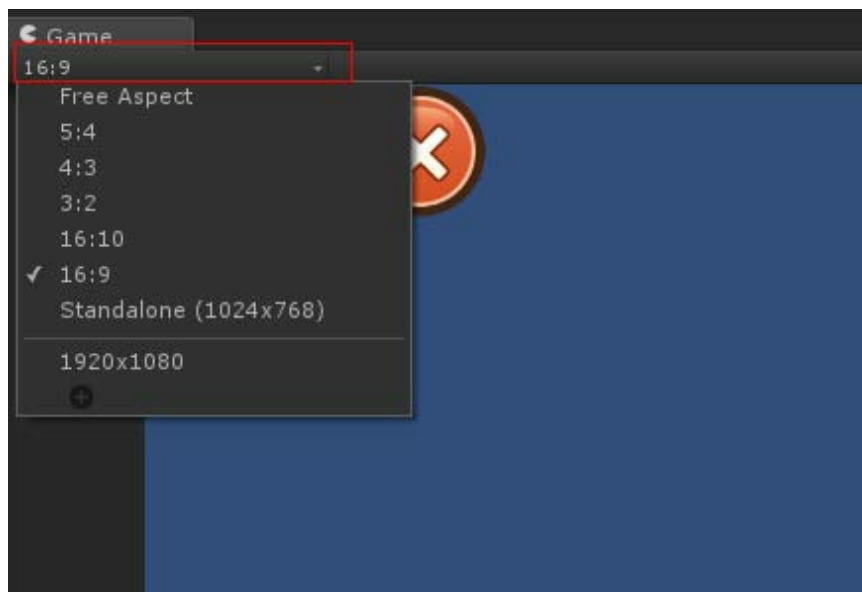


▲ 图6.1



▲ 图6.2

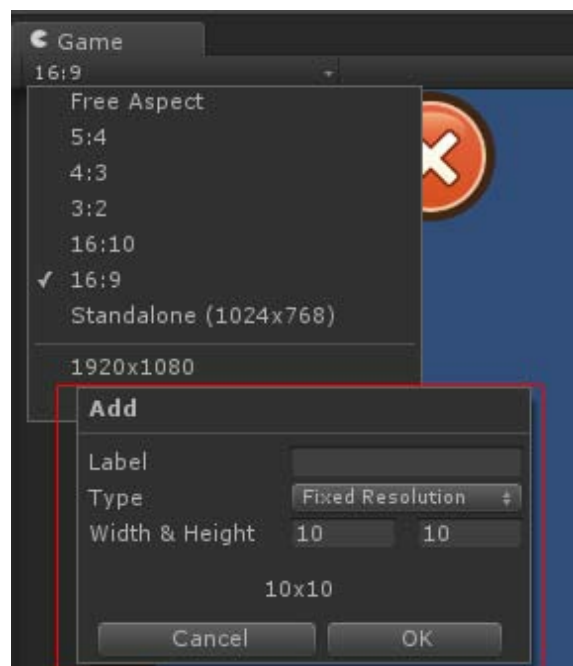
切换屏幕比例模式的方法为单击图6.3所示的位置（Game视图中），会出现图6.3所示的屏幕比例菜单，FreeAspect为不限长宽比，我们可以在其中选择想要的长宽比。



▲ 图6.3

如果屏幕比例菜单中没有想要的屏幕比例，可以单击菜单最底部那个小小的加号按钮，会弹出图6.4所示的界面来让我们自定义一个新的屏幕比例模式，图6.4中的Label选项中，可以输入一个名称来为我们这个自定义的屏幕比例命名。在Type中可以选择FixedResolution和Aspect Ratio 两个选项，如果选择FixedResolution，可以在下面的Width&Height中输入我们的分辨率的宽和高的具体像素；如果选择Aspect Ratio，可以在下面的Width&Height 中输入我们的分辨率的宽高比。设定好之后单击OK按钮，即可保存这个自定义的屏幕分辨率模式。

因为NGUI的UIRoot具备FixSize模式，所以，一般在进行UI随屏幕自适应时，主要着重解决的是屏幕宽高比发生变化之后的自适应。



▲ 图6.4

### 6.1.2 主流设备的屏幕分辨率

作为一个客户端程序，在开发游戏时必须先了解一下市面上主流设备的分辨率和宽高比，这样才能在开发的时候做到有备无患，如表6.1和表6.2所示。

表6.1为IOS设备。

表6.1

设备名	屏幕分辨率	屏幕宽高比
Iphone4	960*640 像素	3:2
Iphone4S	960*640 像素	3:2
Iphone5	1136*640 像素	16:9
Iphone5S	1136*640 像素	16:9
Iphone6	1334*750 像素	16:9
Iphone6 plus	1920*1080 像素	16:9
Ipad2	1024*768 像素	4:3
Ipad3	2048*1536 像素	4:3
Ipad4	2048*1536 像素	4:3
Ipad Air	2048*1536 像素	4:3
Ipad mini 1	1024*768 像素	4:3
Ipad mini 2	2048*1536 像素	4:3

表6.2为Android设备。

因为Android设备品牌型号繁多，所以只列举一部分，大部分Android设备是16:9。

表6.2

设备名	屏幕分辨率	屏幕宽高比
三星 Note2	1280*720 像素	16:9
三星 Note3	1920*1080 像素	16:9
三星 S3	1280*720 像素	16:9
三星 S4	1920*1080 像素	16:9
三星 S5	1920*1080 像素	16:9
小米 3	1920*1080 像素	16:9
小米 4	1920*1080 像素	16:9
魅族 MX3	1800*1080 像素	15:9
华为酷派等其他品牌	一般为 1920*1080 像素	一般为 16:9

至于PC显示器和Mac电脑的屏幕在这里就不一一介绍了，有兴趣的读者或者开发PC/Web游戏的读者可以自行去查一下。

小结：不论是PC设备还是移动设备，屏幕分辨率一般处于4:3到16:9之间，一般来说，我们只需要考虑这两个值作为极限值即可。

### 6.1.3 自适应核心组件Anchor的使用

NGUI作为一款最强大和成熟的插件，自然有非常成熟的屏幕自适应技术，它就是Anchor组件。Anchor组件从旧版本一直更新到新版本，依然是主流的屏幕适配方式。

所谓 Anchor，即为锚点，它的工作原理是它会自动地绑定摄像机的某一个点作为锚点，锚点一共有上左、上、上右、左、中、右、左下、下、右下9个点可以设定。当相机变动时，Anchor组件所在的物体位置也会跟随相机的变动而变动，并始终处于相机边界的锚点位置。

有了Anchor的这个特性，我们一般会在UIRoot下创建Anchor的子物体，例如，我们在UIRoot下创建一个Anchor物体，绑定到相机视窗的左上角，然后将需要始终处于屏幕左上方的UI控件放置于这个Anchor之下，利用Unity中子物体会跟随父物体的变化而变化以保证相对位置不变的特点，实现UI控件随屏幕自适应。

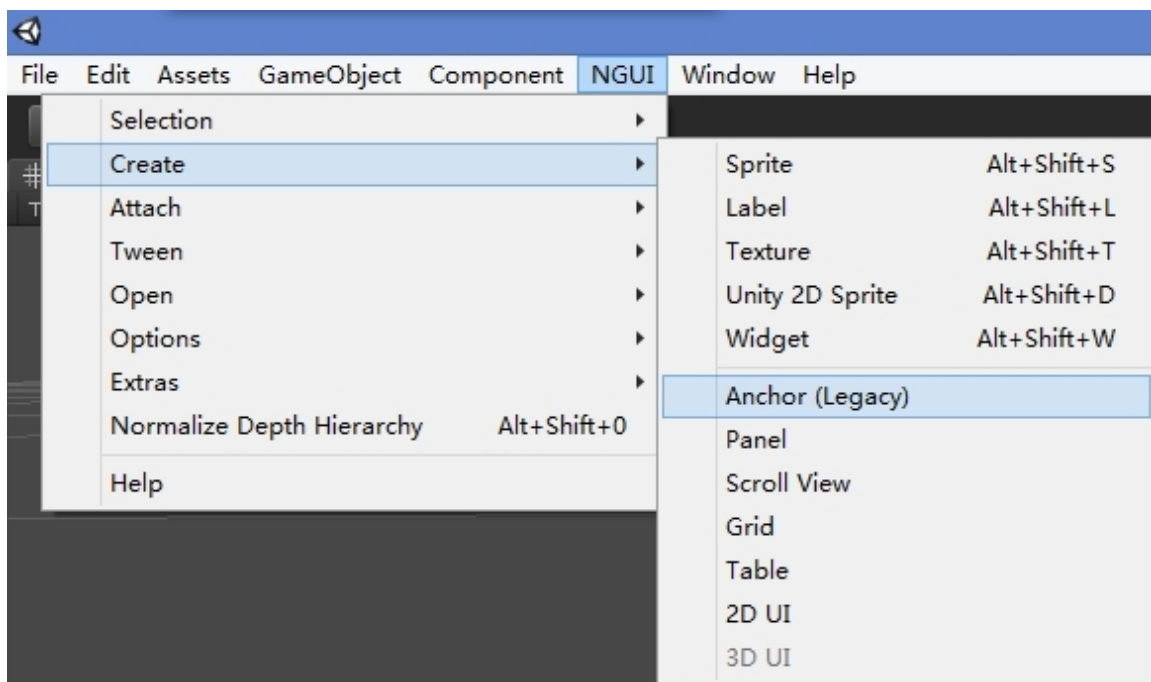
我们先来了解一下Anchor的创建方式，选中要创建Anchor的UI节点，在Unity顶部NGUI菜单中选择Creat → Anchor即可，如图6.5所示。

Anchor组件的设置界面如图6.6所示，我们来了解一下Anchor组件的设置。

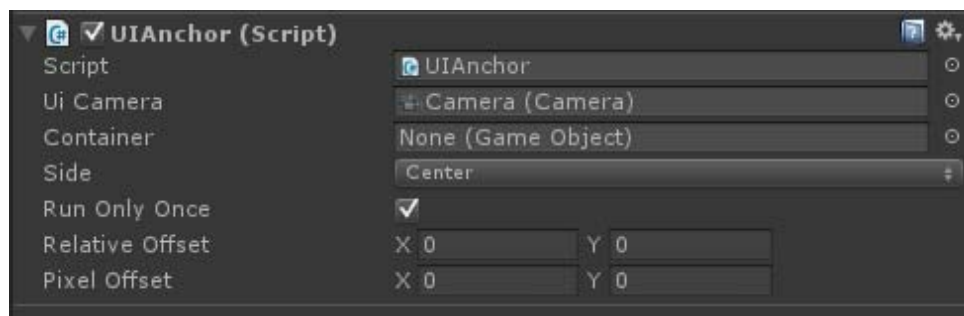
- 从图6.6 中可以看到UICamera 选项自动锁定了该Anchor 所在的UI的摄像机，它将会绑定这个摄像机边缘上的某一个点作为锚点。Container 即为包含的物体，一般情况下无需设置，因为将物体放置于Anchor的子物体中，就可以利用“子物体跟随父物体”实现跟随Anchor的锚点。

- Side 一项为核心项，选择该Anchor 的锚点，一共9 个点，分别对应相机边缘的上左、上、上右、左、中、右、左下、下、右下。

- RunOnlyOnce 意为执行一次，即只在开始的时候进行一次适配，默认为勾选上，一般不需要去修改它。



▲ 图6.5



▲ 图6.6

●**RelativeOffset**，这是Anchor 的相对位置偏移，百分比形式的。比如我们填入X 值0.1，即为Anchor相对于相机边缘上的锚点向X正方向移动10%屏幕宽度的距离。也就是说，如果设定Anchor为Left，但是又将RelativeOffset的X值设为了0.5，就相当于Anchor跑到了中点。

●**PixelOffset**，像素偏移，Anchor 会相对于相机边缘上的锚点以像素为单位进行偏移，例如，我们填入X为100像素，那么Anchor就会朝X方向偏移100像素的距离，这个距离因为是像素单位的不是以屏幕百分比为单位的，所以，分辨率越高的屏幕肉眼看到的偏移量就越小。

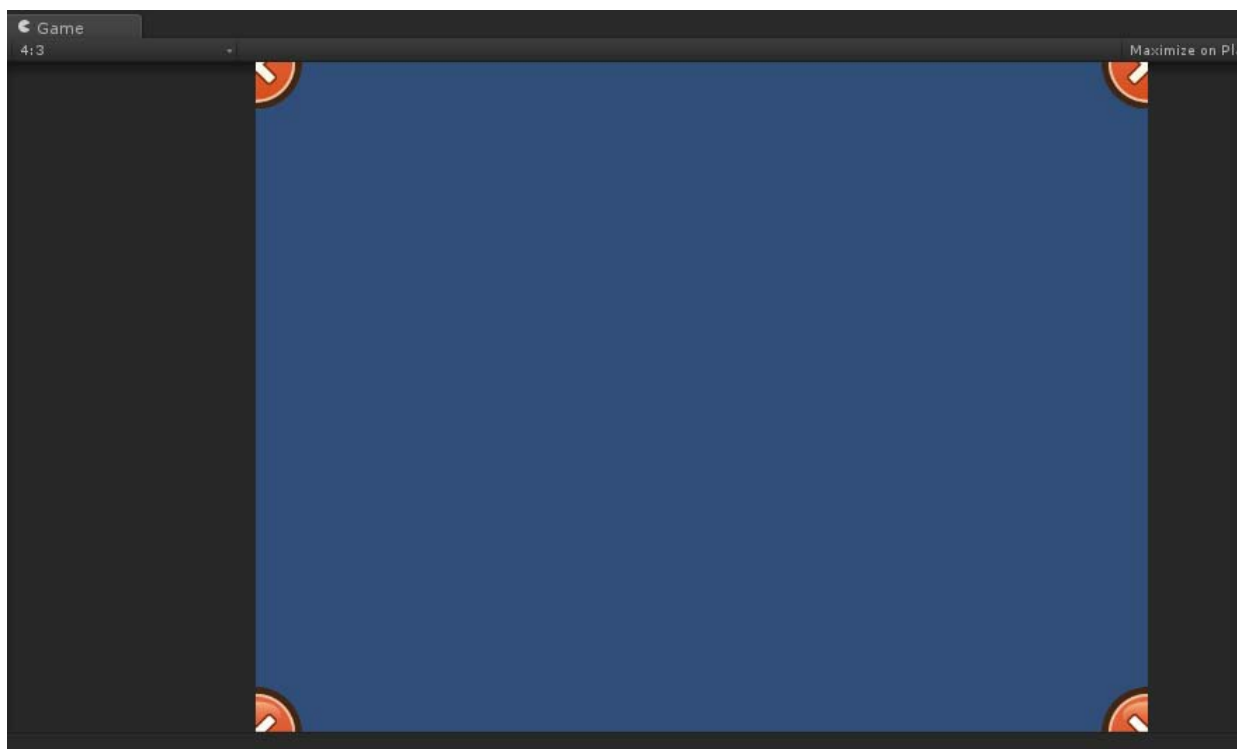
一个简单的自适应案例：



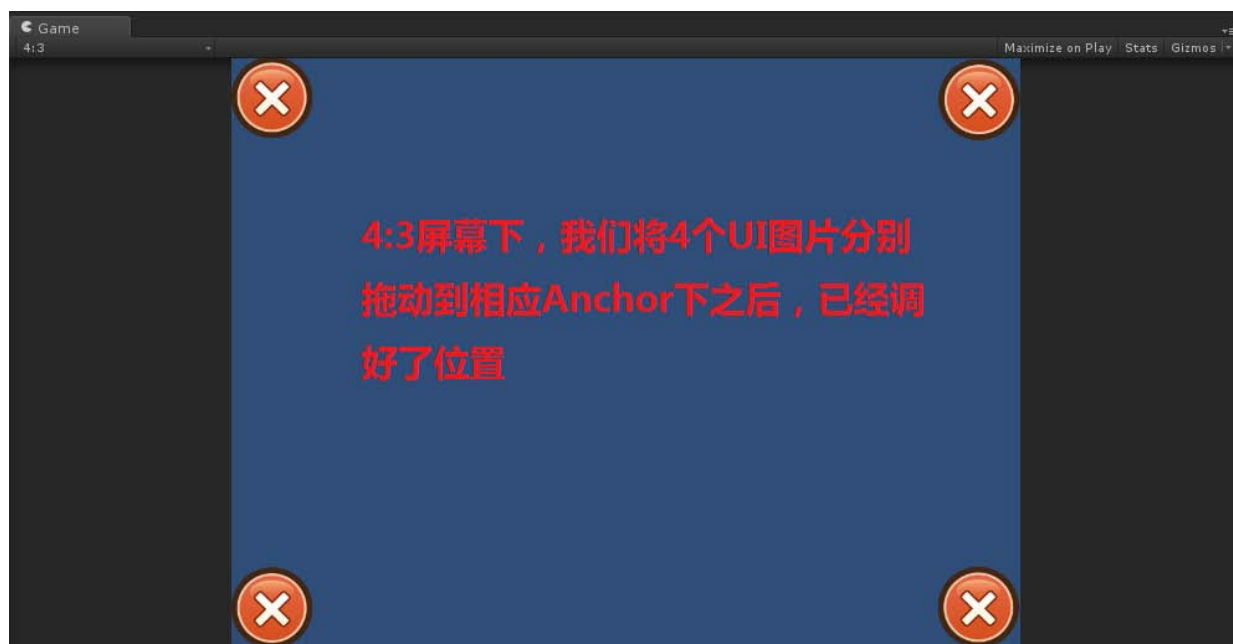
我们现在来简单对图6.1中的4个角的UI图片进行屏幕自适应设定，以加深对Anchor的了解。我们在UIRoot下创建 4个Anchor，分别命名为Anchor\_TopLeft、Anchor\_TopRight、Anchor\_BottomLeft、Anchor\_BottomRight，然后将它们的Anchor组件中的Size分别设为左上、右上、左下、右下4个点。然后我们将4个角的UI图片分别拖到它们所属的Anchor下作为子物体，并Reset它们的transform组件（让它们的本地坐标归零，回到Anchor所在的位置），如图6.7所示。

从图6.7中看出，每个UI图片都只露出了一部分，因为它们的中心点和Anchor所在的锚点重合了。现在可以设定每个UI图片的Transform的Position位置（直接改坐标或者拖动都可），让它们达到我们满意的程度，然后当屏幕宽高比变了后，它们也会跟随Anchor所在的锚点一起发生变化，如图6.8和图6.9所示。

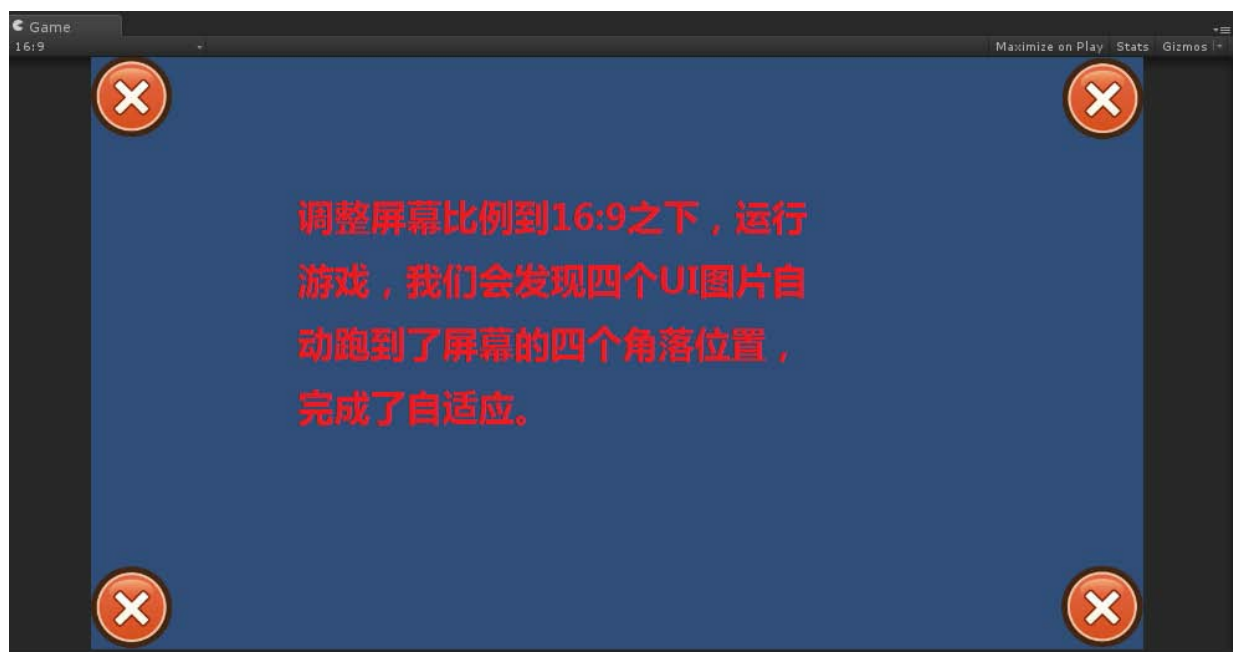
如果我们在调整了屏幕宽高比模式之后，发现UI没有自动适应位置，这是因为Anchor还没有运行，只需要运行一下游戏即可看到UI完成了自适应（确保UI是在Anchor物体下作为子物体存在）。



▲ 图6.7



▲ 图6.8



▲ 图6.9

### 6.1.4 使用Anchor的注意事项

**Anchor** 是开发项目几乎必用的一个非常核心的组件，我们在使用时，需要注意以下一些问题。

(1) 我们在设定了**Anchor**的**Side**锚点之后，**Anchor**会自动跑到相应的锚点位置上去，不需要我们手动拖动**Anchor**。

(2) 不论是3DUI还是2DUI，**Anchor**的用法是一模一样的，不要手动拖动**Anchor**的位置。

(3) 一般情况下，尽量不要去设置 **Anchor** 的 **RelativeOffset** 和 **PixelOffset**，就让 **Anchor**保持锚点原位置，然后将UI控件放到**Anchor**下作为子物体，再去调整UI控件的位置。

(4) **Anchor**物体身上，尽量保证只有**Anchor**一个组件，以便于管理维护。

(5) 一套UI体系中，可以有无数多个**Anchor**（例如有5个**Anchor**都定位于左上角是被允许的），但是，尽量确保**Anchor**的父物体中没有**Anchor**，也就是尽量避免**Anchor**中套**Anchor**。**Anchor**的父物体可以是**UIRoot**，也可以是一个空物体。

(6) 不要滥用**Anchor**，如果相机边缘的9个锚点中的每个点都有多个**Anchor**来定位，那么一定是UI结构的设计有问题了。

### **6.1.5 正式开发UI之前必须明确的几个问题**

我们在正式开发游戏时，适配是必须提前考虑的问题，如果没有考虑好，可能会在项目后期造成大量的麻烦。所以，在正式开发UI之前必须明确如下问题。

(1) 我们针对的是什么样的平台？

(2) 我们游戏的设备屏幕宽高比最大、最小是几比几？

(3) 我们游戏开发时的标准分辨率是多少像素？这将关系到美术制作图片资源的标准。

(4) 检查UI设计草图，和策划人员明确哪些UI会在屏幕宽高比变化时自适应位置？

(5) 明确需要自适应位置的UI分别属于哪一个锚点，并设计一个最佳的UI结构。

## 6.2 UI元素的相对自适应

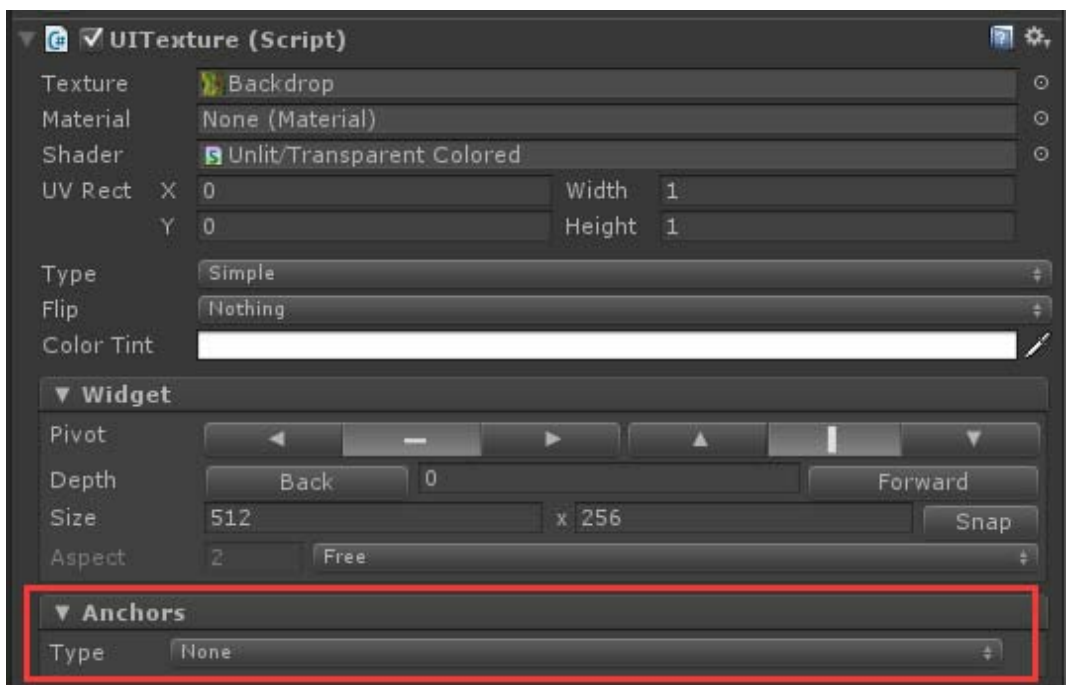
### 6.2.1 什么是UI元素的相对自适应

UI元素的相对自适应，顾名思义，是指两个UI元素之间保持一种相对的位置不要变化，例如，UI元素A永远处于UI元素B右边的50像素处位置。再比如，一个UI背景框，不论屏幕尺寸怎么变化，框的左边缘永远距离屏幕左边 100 像素，框的右边缘永远距离屏幕右边100像素。

### 6.2.2 Anchors的介绍及使用

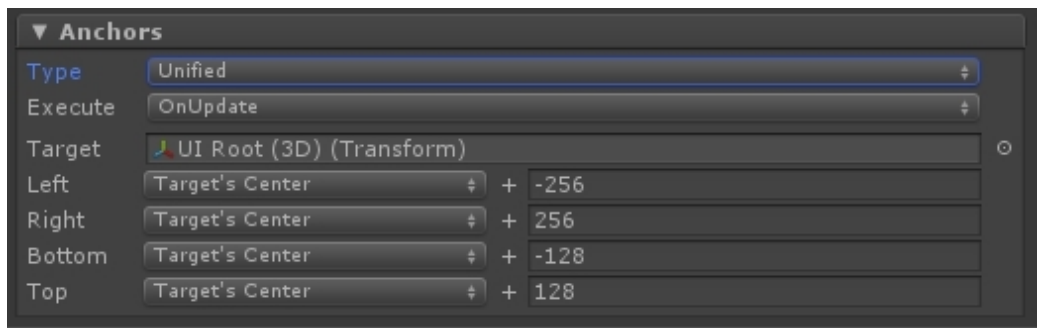
在6.1节中讲到了整个UI随屏幕自适应的问题，使用了一种比较强大的锚点定位组件：Anchor，这里将要学习NGUI新版本中提供的一种全新的定位工具：单个UI控件的Anchors单独定位。

Anchor的主要作用是一个组件，它将会自动去寻找摄像机的边缘和中心点一共9个点，以此作为基准点进行定位适配。而Anchors（为了区分，以下皆用相对Anchors来形容）是每一个UI控件都会附带的选项，它能为每一个UI控件设定绑定对象和绑定的方式。相对Anchors的界面如图6.10所示，默认情况下为none，意味着控件默认情况下无相对定位。



▲ 图6.10

我们来了解一下这个相对Anchors选项的工作原理。首先，选择控件的Anchors的Type为Unified，会看到如图6.11所示的界面。



▲ 图6.11

在界面中，Type为Anchors定位的类型，一共有None（默认状态，无相对定位）、Unified（统一的标准定位）、Advanced（高级定位）。统一的标准定位和高级定位之间的区别在于，高级定位能够有更加复杂的定位设置，但它们的定位原理是一样的。

Execute选项为设定相对定位执行的时机，一共有3个选项可以选择：OnUpdate、OnEnable和OnStart，我们知道Update、Start、

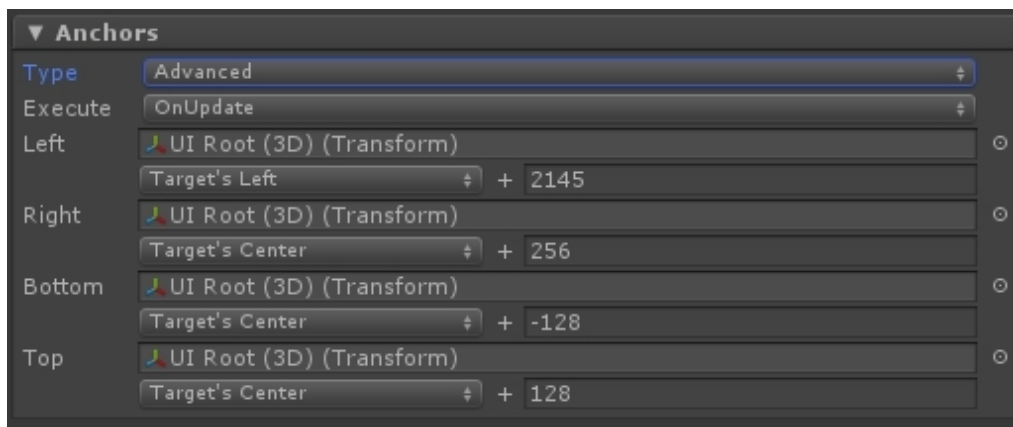
**OnEnable**是Unity中脚本自动执行的函数，如果选择了**OnUpdate**，则这个控件的定位会随着每一帧去进行更新，如果选择了 **OnEnable**，那么意味着这个控件只会在激活的时候更新一次定位，然后不再更新，直到下次激活再进行更新，如果选择了 **OnStart**，那么这个控件在场景加载之后只会更新一次相对定位信息。一般来说，为了节省性能又照顾需求，我们会选择**OnEnable**。如果有特殊需求，例如，我们在打开一个界面时，可以拖动界面的4个角来改变界面的尺寸，而界面中的某个元素又需要实时地和改变之后的界面进行相对位置的匹配，就需要选择**OnUpdate**，它会每一帧都去更新相对位置的信息。

**Target**选项为这个控件相对位置选取的参照物，也就是这个控件以哪一个物体作为锚点来进行相对位置的绑定。

下面可以看到有 4 个选项：**Left**、**Right**、**Bottom**、**Top**。它们的作用是设置这个控件的 4 个边的对位信息，我们在每个选项旁边的下拉菜单中选择**Target's Center**（以目标点的中心点作为参照）等目标体身上的参照点，最右边的数字表示的是像素偏移，加号为向 **X** 轴正向偏移一定像素，减号为向**X**轴负向参照目标点偏移一定像素。

从上文的介绍中可以看出，**Anchors**适配的核心原理为：为该控件的4条边分别设定相对位置的参照点，这个参照点可以选择参照物体身上的中点、左边点等，然后设定一定量的偏移像素。这样当相对**Anchors**执行时，控件的4条边会分别按照定位信息朝4个方向“拉扯控件”，来达到相对位置适配的效果。所以，**Anchors**本质上是点对点的（自身控件的4条边的中点，分别去对应一个参照点）。

图6.12所示为**Anchors**的**Advance**模式下的设置界面，可以对比**Unified**模式的设置界面，很容易发现**Advance**模式只是比**Unified**多了一些功能，可以给控件的4边分别设置不同的参照物。



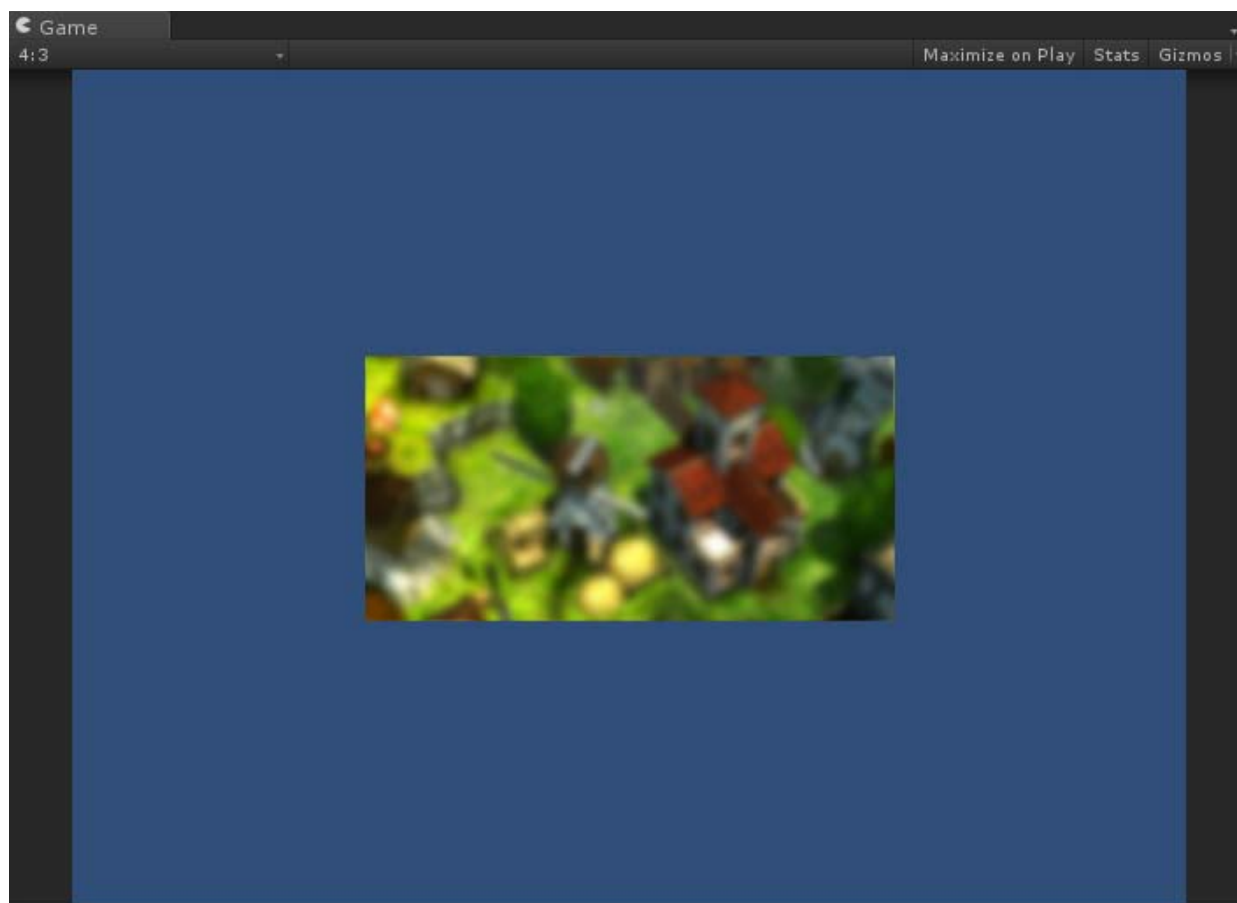
▲ 图6.12

### 6.2.3 使用Anchors的范例：背景图的全屏适配

下面我们来做一个Anchors适配的范例：让一张背景图，无论在什么屏幕比例下面，都能刚好充满全屏（即使被拉伸）。

如图6.13所示，我们在场景中添加了一张Texture。





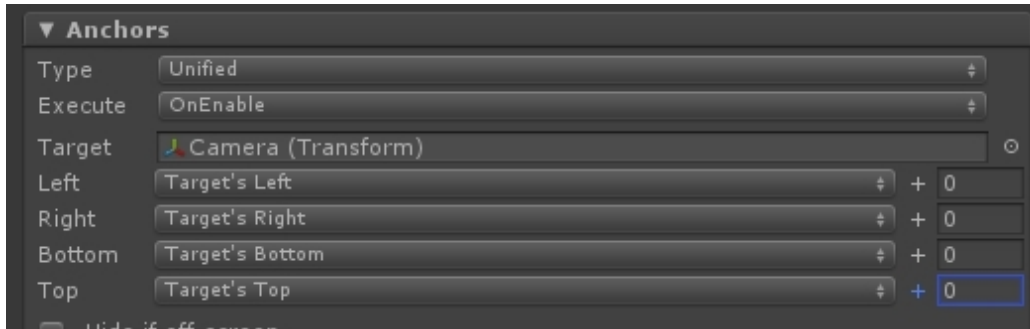
▲ 图6.13

在它的Anchors设置选项中，首先选择Unified标准模式，然后选择执行方式为OnEnable的时候执行一次。

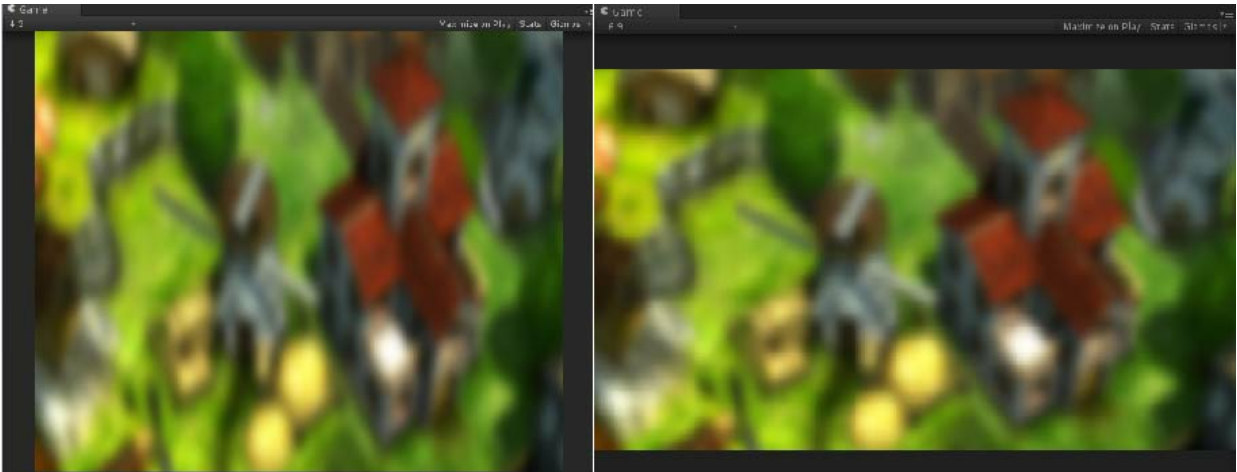
既然希望这张图永远充满全屏幕，屏幕就是相机的视窗，当屏幕比例发生变化的时候，本质上是相机的视窗大小发生了变化了，所以，这里我们相对定位的参照物为渲染这个 UI 的摄像机。

因为屏幕长宽比变化的时候，会根据高度进行相机的长度变化（4:3变化到16:9其实效果类似于屏幕被“拉长”了）。所以，我们设置这个Texture控件的左边定位点为目标的左边，右边定位点为目标的右边，上下两边的定位点为目标的目标的中心点即可。然后将像素偏移全部设为0，这样就让控件的4个边全部被拉到和它们的锚点重叠，造成全屏效果。

如图6.14是我们对Texture组件的Anchors的设置，图6.15是完成之后运行游戏的效果，在4:3和16:9的情况下，背景图都刚好充满了全屏，但是，在进行相对点定位的过程中，导致了图片发生了形变。



▲ 图6.14



▲ 图6.15

### 6.2.4 使用Anchors的注意事项

因为存在Anchor组件和每个控件的相对Anchors两种定位方式，所以，使用相对Anchors时，我们一定要注意以下一些事项。

(1) Anchors需要对每个控件都进行详细的锚点定位设置，工作量巨大，所以，在没有必要的情况下，尽量用Anchor组件，可以减少很多工作量。

(2) 使用**Anchors**时一定要切记它在屏幕分辨率变化的情况下，很可能会导致控件形变（控件4边被拉发生变形）。

(3) 如果一定要使用**Anchors**，又不希望它形变，就将**Anchors**的4边定位都以目标物体的同一个点（如**Target's Center**）作为参照点。

(4) 使用**Anchors**来进行相对定位，一定要深刻熟悉它的原理，并谨慎设置，以免产生不希望见到的Bug。

## **6.3 多摄像机同时协作运行**

### **6.3.1 摄像的渲染层的概念**

Unity 中，每一个物体都有一个所处的“层”的概念，也就是物体的**Layer**，而摄像机中可以通过设置**CullingMask**来决定该摄像机只渲染哪些层的物体。对于UI来说，这个原理也一样适用，因为NGUI的每一个控件元素，本质上都属于一个**GameObject**，都有自己的层；每一个**UIRoot**都会自带一个**UICamera**来渲染UI。

利用这个原理，我们可以实现多个摄像机来进行UI的渲染，例如，场景里可以放置一个3D摄像机负责渲染3DUI，还可以放置一个2D摄像机，负责渲染2DUI等。理论上来说，可以放置无限多个摄像机来分别渲染不同层的内容。

如图6.16所示，当场景中有两个摄像机都会渲染同一个层的时候，会造成画面重复显示，图6.16中的**Texture**就被显示了两次。



▲ 图6.16

### 6.3.2 多摄像机协作的应用范围

多摄像机协作应用，是为了对付一些比较复杂的UI情况，因为Unity中很多元素不适合和UI控件放在一起进行显示，比如粒子、比如3D的模型。例如，在UI中显示一个3D模型，一般就会使用一个UI摄像机来负责渲染UI控件，用另一个摄像机来渲染3D模型，然后让这两个摄像机渲染的内容进行叠加，呈现出一幅完整的UI图像。

图6.17中所示的UI，中间的人物是一个3D模型，由一个独立的摄像机进行渲染；周围的装备栏位和背包等UI元素，又由另一个独立的UI摄像机进行渲染。



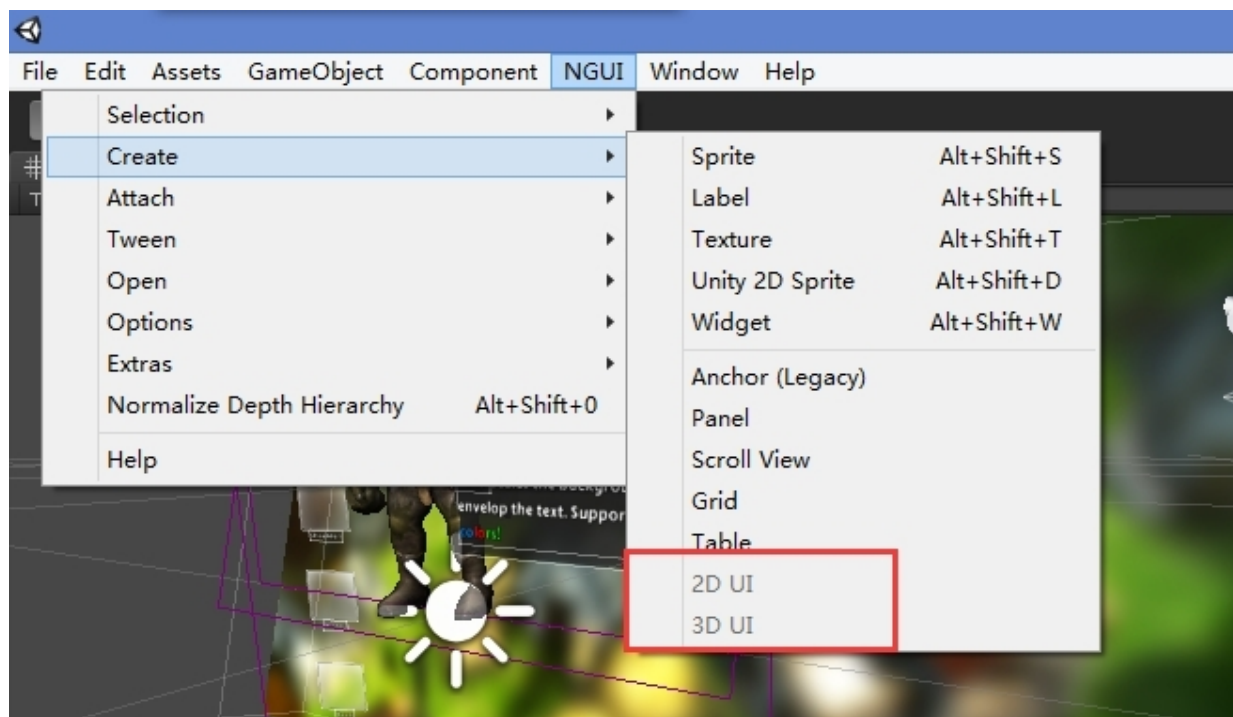
▲ 图6.17

### 6.3.3 如何创建多个UI摄像机

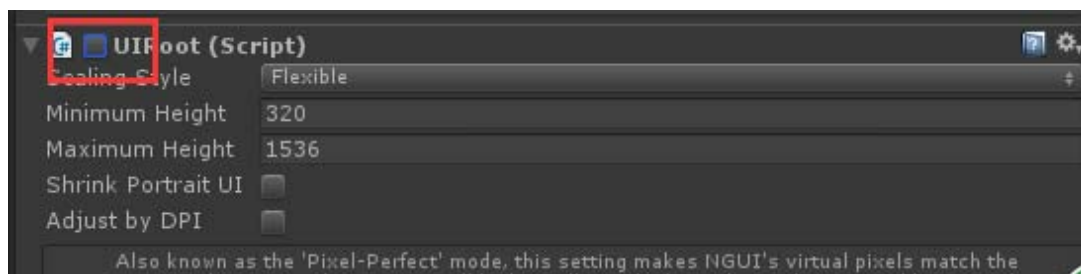
在NGUI中，如果创建了一个UIRoot，当我们再点开Unity顶部的NGUI菜单，企图通过Creat菜单去创建一个新的UI时，会发现创建UI的选项已经变成灰色，无法再创建，如图6.18所示，这是因为NGUI默认只允许场景中出现一个UIRoot。

我们要创建多个UI摄像机时，首先得考虑是否需要UIRoot，如果有UI图片做成的控件，需要被UIRoot根据屏幕自动进行放缩，那么就需要一个UIRoot来管理这些UI控件。

为了创建多个UIRoot（每个UIRoot会自带一个UI摄像机），可以关闭场景中已经存在的UIRoot物体的UIRoot组件，然后再单击Unity顶部NGUI菜单通过Creat创建，如图6.19所示。



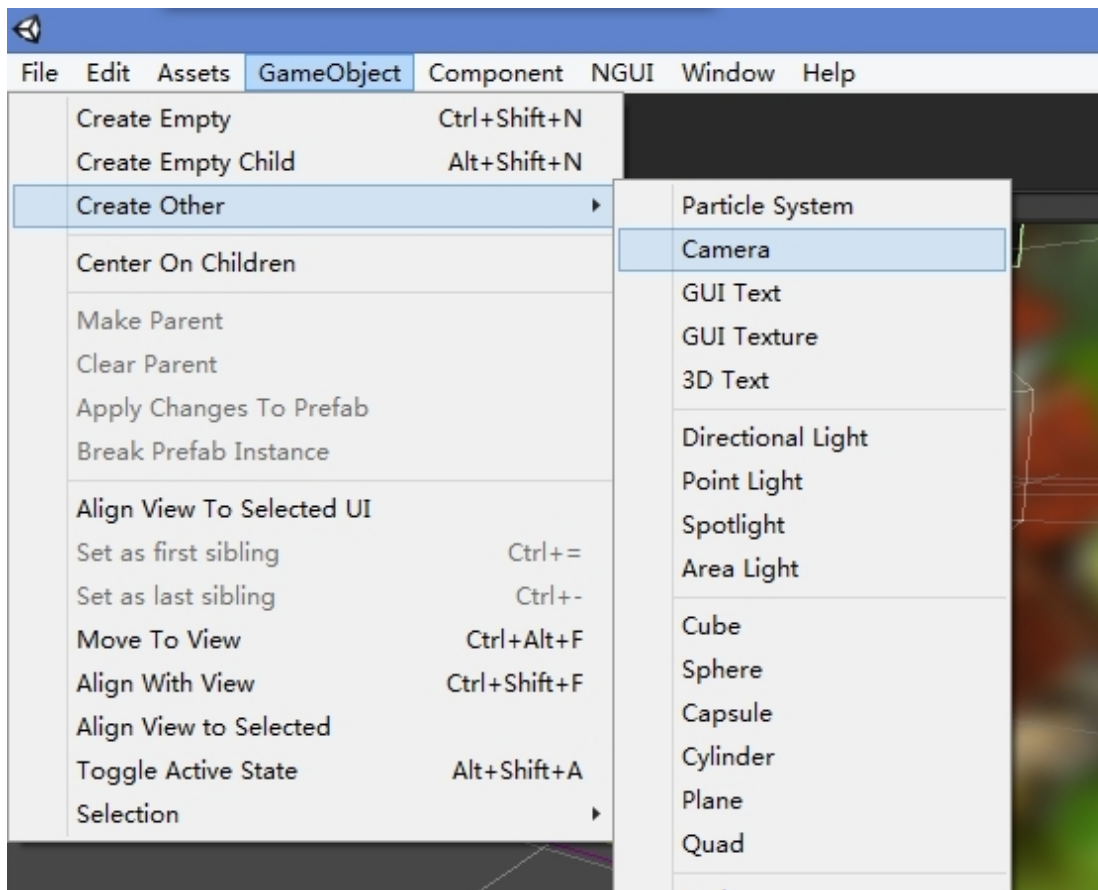
▲ 图6.18



▲ 图6.19

如果不需要多个UIRoot，只需要一个新的UICamera，那么可以在场景中新创建一个摄像机，创建方式为，在 Unity 顶部菜单单击 GameObject，选择 Creat 创建一个，如图6.20所示。





▲ 图6.20

然后在这个新创建的摄像机上附上一个 **UICamera** 组件即可，组件附加方法可以在Inspector面板中依次单击 **AddComponent** → **NGUI** → **EventSystem**（**UICamera**）。如果它渲染的物体不需要接收UI的事件（如单击拖曳等），这个**UICamera**组件就没有必要附上。

### 6.3.4 多摄像机协作的注意事项

因为摄像机直接关系到画面的渲染显示，所以，多个摄像机协作时，一定要注意以下事项。

- （1）规划并设置好每个物体的**Layer**，不要混乱。



(2) 设置好每个摄像机的 **CullingMask**，确保场景中的摄像机之间不会重复渲染同一个层。

(3) 对于多个**UIRoot**的情况，更要检查它们所属的层和该**UIRoot**渲染的层是否对应。

(4) 尽量不要滥用多摄像机协作。

## **6.4 巧用九宫格以减少UI资源量**

### **6.4.1 项目安装包大小对项目的影响**

我们制作游戏完成后，最终发布时都会生成一个发布的“安装包”。对于**PC**来说，它是一个客户端安装包，对于**Web**来说，它是一个网络在线下载的资源包，对于**安卓**和**苹果**平台来说，它是一个**APK**包或者**IPA**包，不管怎样，最终都会有一个“打包资源并发布”的过程。

如果一个游戏的安装包过大，会导致用户成本急剧升高。每一个游戏在招揽玩家时都会付出一定的成本，比如广告推广、运营活动等，如果用户成本过高，会导致付出同样运营成本的情况下，游戏的玩家会更少。在这个用户成本的价值体系中，安装包大小占了很重要的一部分因素。拿**安卓**平台做例子，一个**50MB**大小的游戏，和一个**200MB**大小的游戏，它们在付出同等运营资源的情况下，**50MB**的游戏获取的用户数会大大超过**200MB**的游戏，虽然这过程会受游戏品质、题材、美术风格等的影响，但是，总体上包大会吃很大的亏。

由此可见，对于客户端程序来说，资源的整合和优化，让项目的资源量尽可能变少是对整个游戏有巨大帮助的。

### **6.4.2 UI资源量对资源包大小和内存的影响**

UI资源在这里单独拿出来分析，一是因为我们主要讲解UI的制作开发，二是因为UI资源在游戏项目中相对比较特殊，它具有以下特点。

(1) UI资源几乎都是图片，而图片是最占资源量的资源类型之一。

(2) Unity不支持外部压缩，即使在外部分将一个10MB的图片压缩到只剩1MB，导入引擎之后它也会被解压出来变成10MB，然后转换为各平台特有的格式。

(3) UI资源一般都贴着相机视窗显示在最上层，也是游戏中人机交互的重要部分，一般需要保证高质量，所以一般都使用RGBA32位色（真彩色），这会让资源量更大。

(4) UI资源一般涉及大量图标、背景框等，如果要保证高分辨率，它们将会导致资源量很大。例如，为1080P的屏幕做一个全屏的背景框，就导致一个背景框就是一张1920\*1080的图片，这种资源量大，我们必须想办法进行优化。

而且因为UI资源几乎都是以图集形式展现，所以在加载的时候，它会被整个一块儿加载进内容，如果不优化好UI资源量，会增大内存的开销。

综上，我们可以知道，UI资源虽然不一定是项目中最大头的资源部分，但UI资源一定是项目开发中必须优化好的一块“重地”。

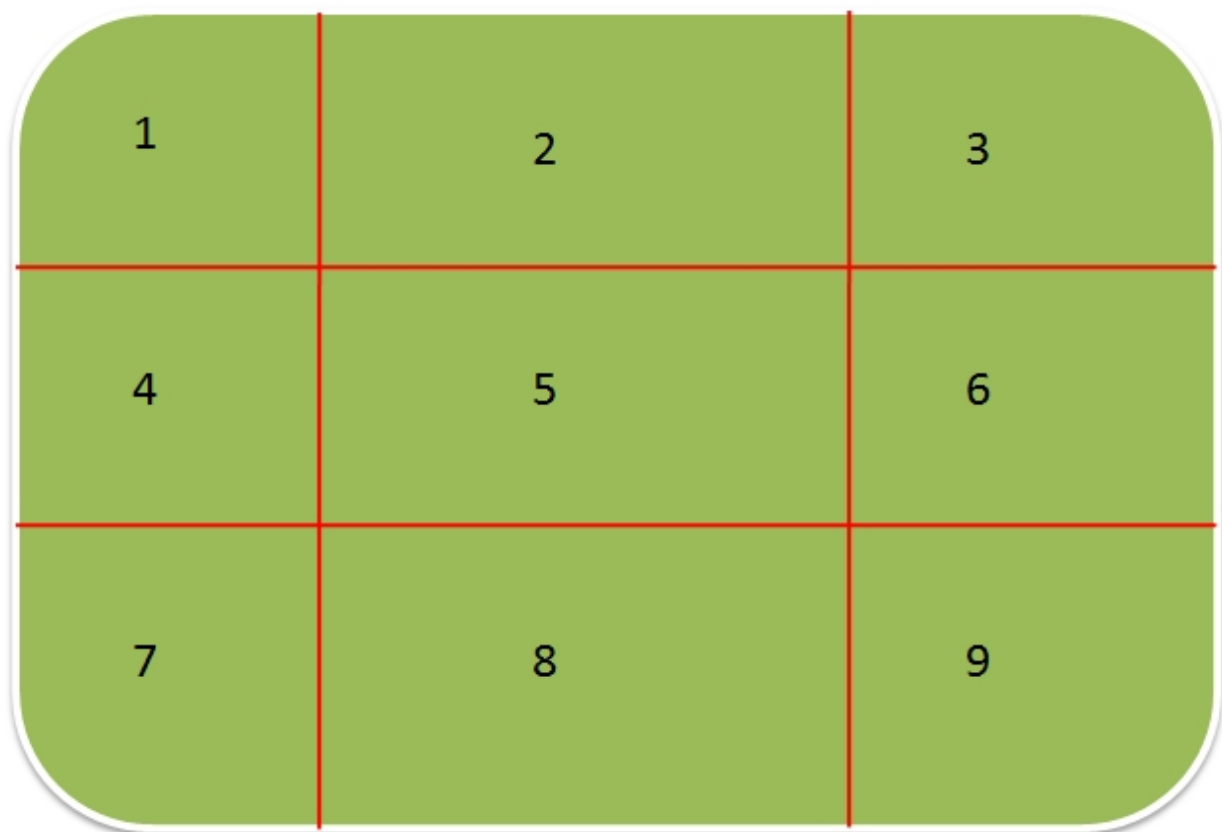
### 6.4.3 什么是九宫格UI

九宫格，顾名思义，就是9个格子，是一种做UI常见的方式，来增强UI资源的复用性并减少资源量。九宫格的主要目的是处理图片拉伸效果，我们知道图片一旦被拉伸，它就会出现形变、模糊等问题，但是，有的图片它的某一些部分又是允许被拉伸的。例如，一个UI背景

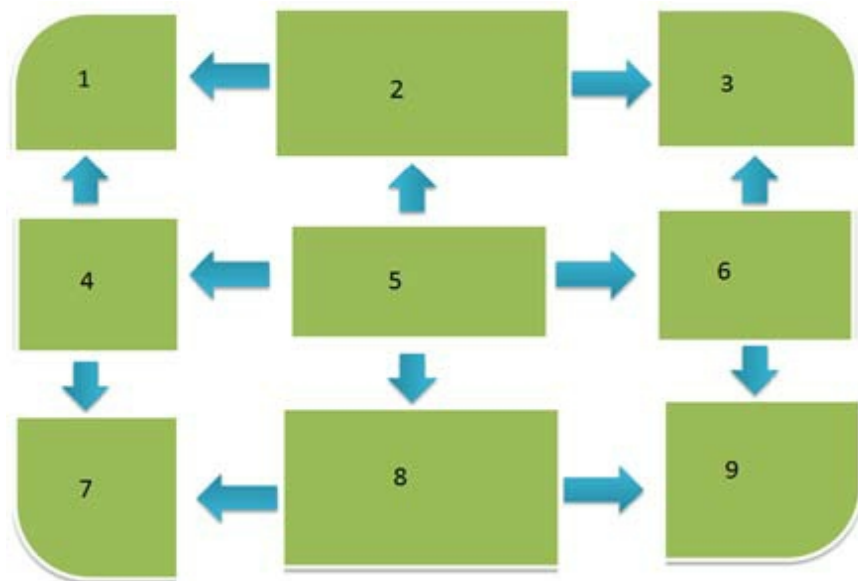
框，它的中间部分几乎是一个纯色，允许被拉伸（纯色被拉伸不会发生质量问题），但是，边缘的4个角可能有一些特殊花纹或者倒角不允许被任意拉伸，这个时候就可以用九宫格，来使4个角不进行拉伸放大，只让中间部分进行拉伸放大，达到一个小框拉大成一个大背景框使用的同时图片质量又不会发生损伤的目的。

如图6.21所示，我们将一个UI背景板分割成了9块，像一个九宫格一样，这9个格子并不要求一定要相同大小。

当我们拉伸放大的时候，1、3、7、9四个角落的格子不会被放大，而2号格子会被左右拉伸以填充1和3之间的缝隙，4号格子会被上下拉伸填充1和7之间的空隙，5号格子会被整体拉伸来填充图片被拉大之后中间的空间。拉伸原理如图6.22所示。



▲ 图6.21



▲ 图6.22

#### 6.4.4 如何让美术提供合适的九宫格UI资源

如果我们需要一个 UI 背景底框，尺寸是 1920\*1080，这个时候如果美术真的提交一个1920\*1080的底框过来，会导致资源量极其庞大，几乎一个底框就占满了一个大图集。所以，我们得让美术尽量地提供九宫格资源，主要有以下一些原则。

(1) 不到万不得已的情况下，对于这种大面积底板性质的UI，尽量只在4个角上做文章，其他地方用可用于拉伸的纯色。

(2) 凡是左右可以缩短的UI、上下可以缩短的UI，都尽量使用九宫格切短了交给程序来做。九宫格不一定是9个格子，也可以是3个格子等。

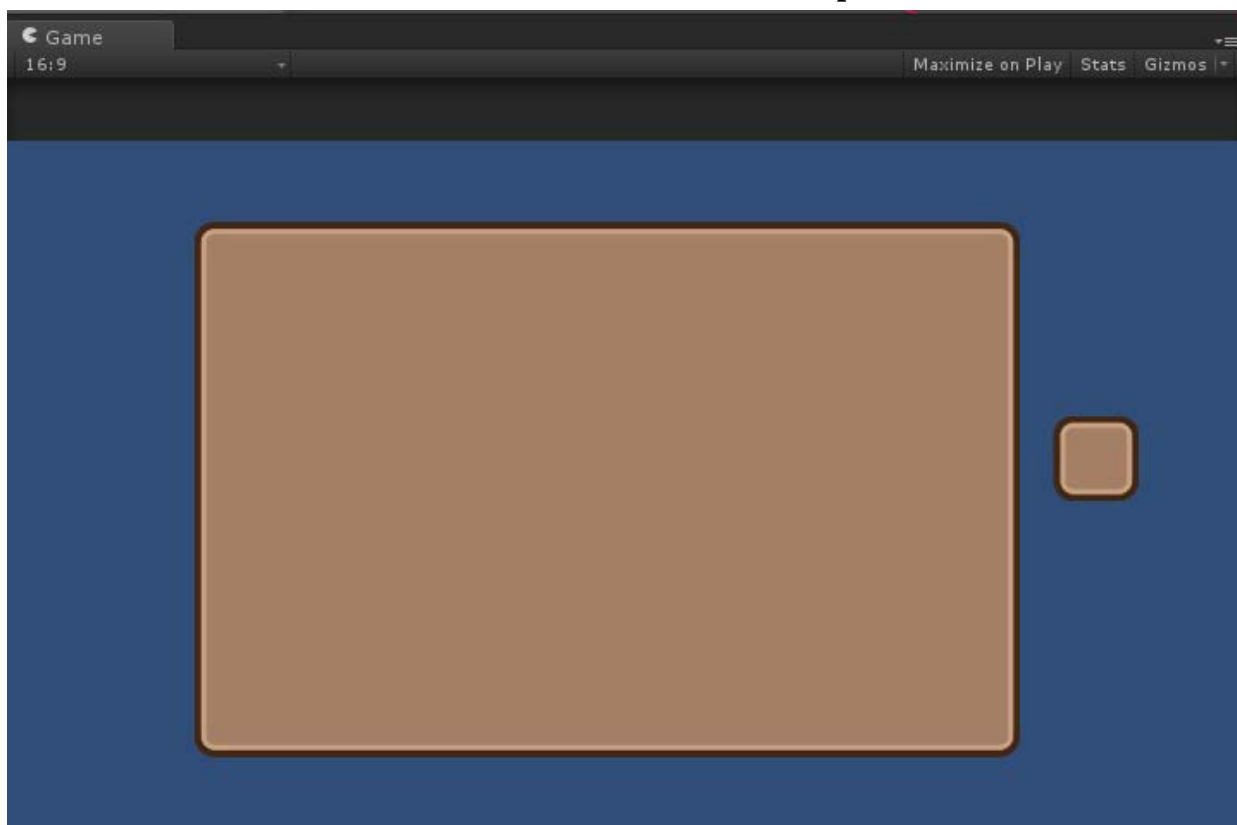
如图6.23所示的大底框，它的资源图片只是它右边的那个小框而已，我们通过九宫格的原理，让它拉大到了一个巨大的程度（可以看到它拉大之后边框线条的粗细并没有被拉粗，4个角的倒角也保持了美术原设计的弧度，这就是九宫格原理起的作用），它依然没有发生分辨率降低等质量问题，但是，资源量却省下了数十倍、上百倍。

### 6.4.5 如何在NGUI中划分九宫格

当美术将一个资源图片合并成一个非常精简小巧的图片交给客户端程序后，客户端程序需要对这个图片进行切割九宫格。切割方法为如下。

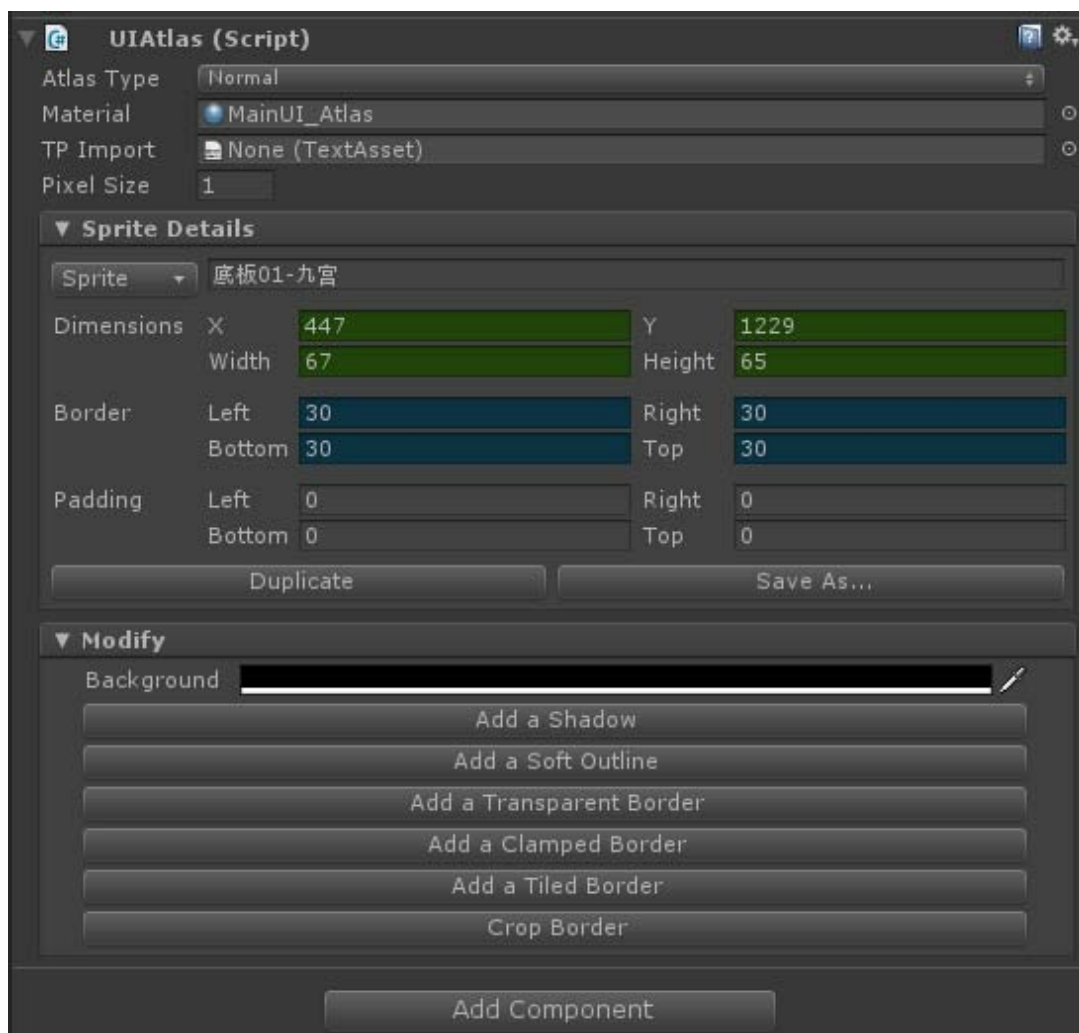
(1) 将这个UI原件制作到图集中去。

(2) 选中图集的 **Atlas** 预设文件（制作好图集后自动生成的那 3 个文件中的预设文件），出现如图6.24所示的Inspector界面。

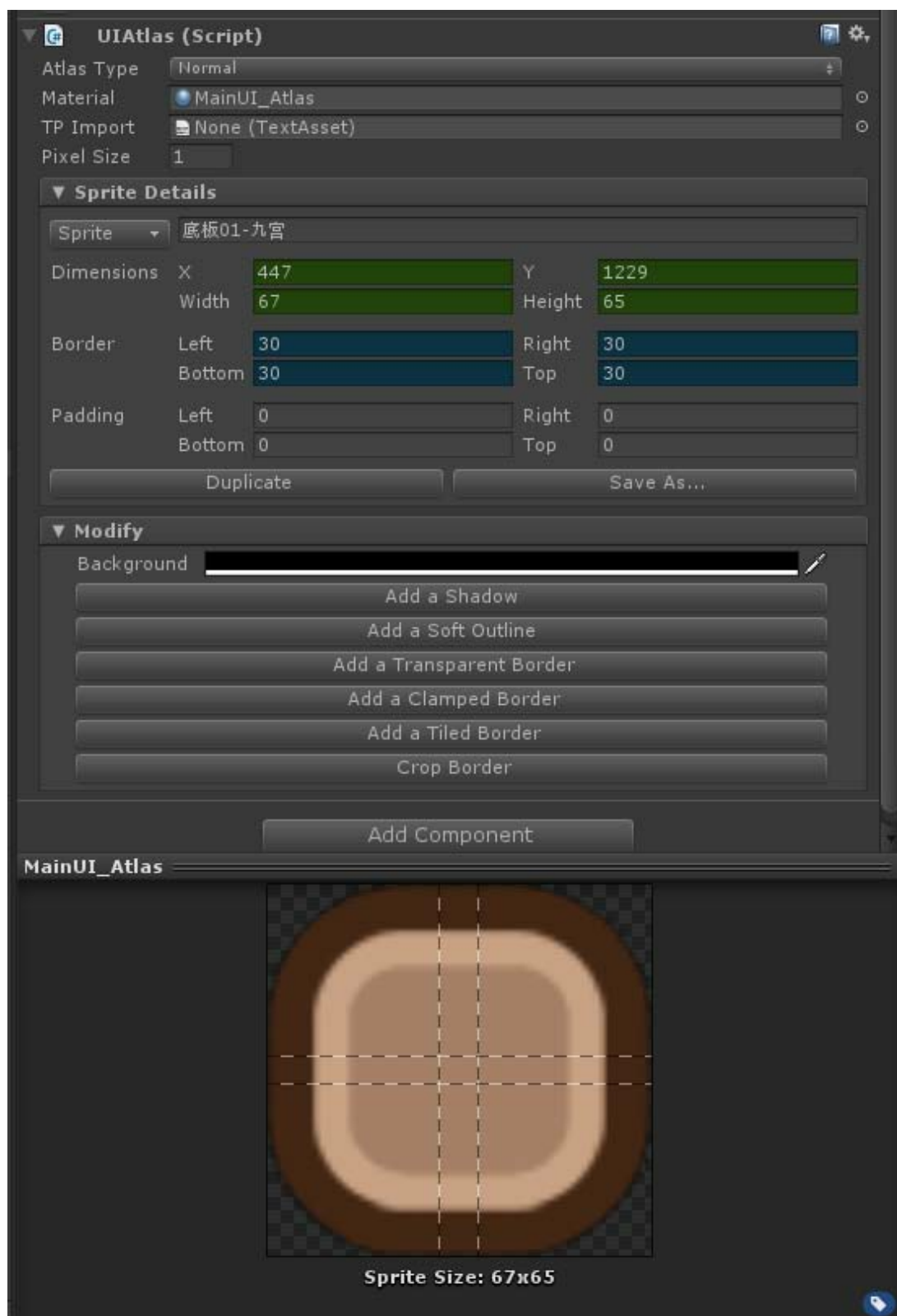


▲ 图6.23

(3) 我们在 **Sprite Details** 中单击 **Sprite** 按钮，会弹出这个图集中所有精灵的预览图，从中选中要切割九宫格的那个 **Sprite**。我们会看到图 6.25 所示的画面，在 6.25 图中的下部会出现这个 **Sprite** 的预览图。



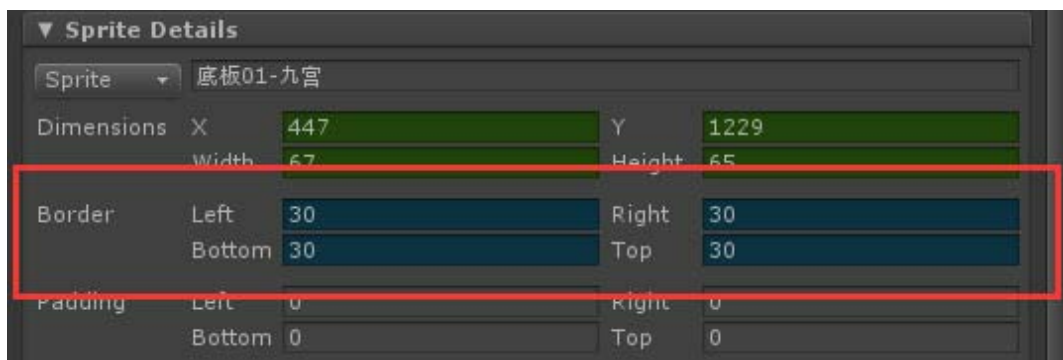
▲ 图6.24



▲ 图6.25

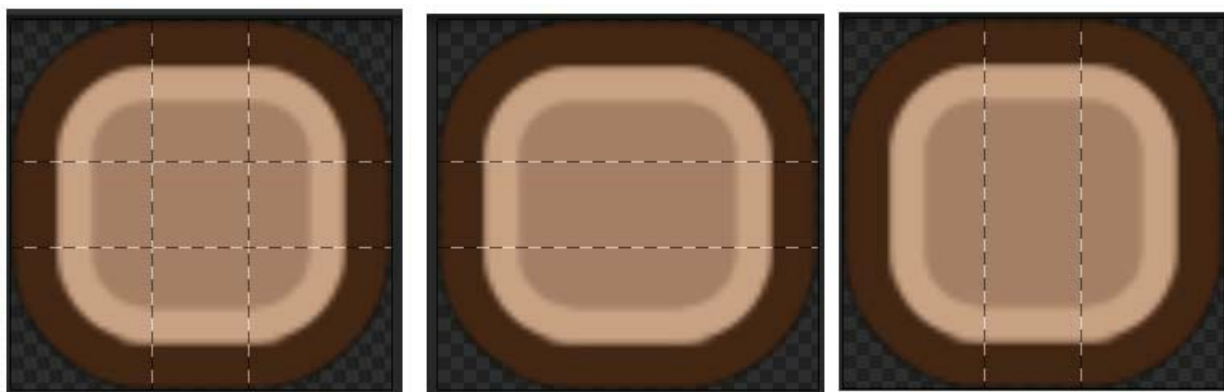


(4) 我们调整图6.26所示的Border设置项，这个Border就是切割九宫格所需要的设置。它一共需要设置4个参数：Left、Right、Bottom、Top，这4个项代表着：从左边朝右X像素位置处划一条竖着的线，从右边朝左X像素位置处划一条竖着的线，从下边朝上X像素处划一条横着的线，从上边朝下X像素处划一条横着的线。这样4条线就将一个Sprite切成了九宫格。



▲ 图6.26

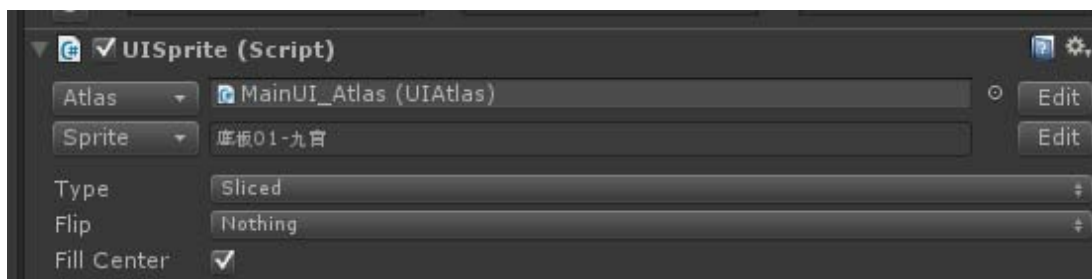
如图6.27所示，展示了几种九宫格的切法。它们从左到右分别是：可以全方位拉伸的九宫格、只能上下拉伸的三宫格、只能左右拉伸的三宫格。



▲ 图6.27

## 6.4.6 如何使用九宫格UI

使用九宫格UI的时候，必须保证这个Sprite是已经按照上一小节的步骤切好了九宫格。然后创建这个Sprite，将它的模式设置为Sliced即可，如图6.28所示。



▲ 图6.28

Fill Center 选项是自动拉伸九宫中的中心格子来充满中心被拉大的区域，如果不勾选 Fill Center 选项，则九宫格被拉伸之后，中心格子不会自动充满，会出现一个空隙。如图 6.29 所示，左边为勾选之后的情况，右边为不勾选的情况。



▲ 图6.29

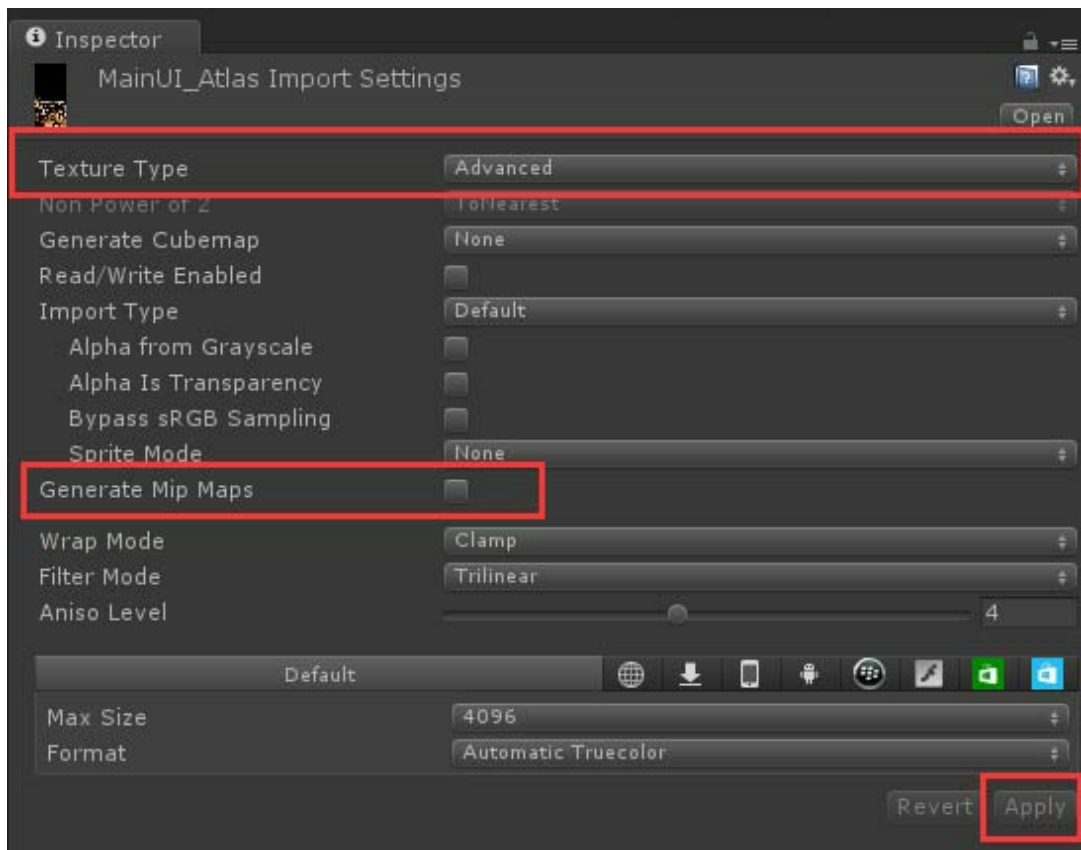
我们也可以将图片设置为Advance高级模式，高级模式的本质区别是，允许对划分的九宫格的每一块设置一个Type，比如左上角的格子平铺、中心格子拉伸等。

### [6.4.7 去掉Mipmap以进一步降低资源包大小和内存占用](#)

首先我们需要了解什么是MipMap。

在Unity中，MipMap纹理图片的一个属性，有开启和关闭两种状态。比如说一张1024\*1024的纹理图片，在导入引擎之后它的默认状态下MipMap是开启状态的，这会导致它会自动生成512\*512、256\*256、128\*128、64\*64、32\*32的这么多张额外的图片。主要用途是，当这个纹理离摄像机足够远时（例如一个人跑到了很远的地方去），这个纹理图片在游戏摄像机中就会很小，这个时候为了节省性能和达到更好的显示效果，Unity会自动选用之前自动生成的众多MipMap小纹理中的一张来替代最初的纹理。

NGUI生成的UI图集也是一张纹理图片，它的默认状态下也是选中了MipMap的，但是，UI是附着在摄像机视窗最上层的，几乎不会存在像人物一样跑到很远的地方去的情况。所以，我们可以选中UI图集的图片文件，在图片类型中选择Advanced高级类型，然后取消掉MipMap的选项，再Apply应用保存即可，如图6.30所示。



## 6.5 实战开发中UI资源制作标准

### 6.5.1 为什么要设定UI资源制作标准

在开发一个游戏项目时，我们应该在项目初期就制定好UI资源的制作标准。这套制作标准并不是指UI应该用什么美术风格，而是UI资源的分辨率标准、输出格式，以及客户端程序和美术人员的“约定”。

比如说，我们也许会在项目开始初期，定好所有的UI都将以1280\*768的画布进行制作并输出，输出的所有格式全部统一为PNG等，我们也可以和美术人员约定UI资源如何分类、如何对接等。

如果忽视了UI资源的制作标准，那么将会给项目埋下巨大的隐患，例如，不同界面的UI分辨率标准不同、UI切割得很乱导致一大堆重复资源浪费了资源包空间。

### 6.5.2 资源制作标准设定建议

一般来说，如果我们使用Unity和NGUI搭配起来开发一个游戏的客户端，可以为项目提出一些资源制作标准的设定技巧。

#### 1. 所有的UI资源全部采用PNG导出

因为Unity不支持外部压缩，所以，不论使用PNG还是JPG，只要尺寸相同，资源量在引擎中都会是一样大。所以，可以大胆地采用PNG进行输出，以保留和实现更好的色彩效果。

#### 2. 设定好一个客户端的标准分辨率

这将会影响美术人员制作图片，我们在开发游戏时有义务提前考虑好这个游戏的客户端是面向什么样的分辨率的设备，是1920\*1080还

是1280\*768?

### 3. 提前考虑是否需要跨平台

如果需要跨平台，那么意味着不同平台的设备之间可能会有屏幕长宽比不同的问题，这在设计人员设计UI布局时会受到很大的影响。

### 4. 和美术人员约定非常大的图片尽可能采用九宫格

在上面已经讲过了九宫格的优势和重要性，而美术人员在设计和制作UI时，一般不会考虑到程序人员会怎么实现，也不会考虑到资源量大小的控制问题。所以，在开发项目时，我们应该和美术人员沟通，让其在设计和制作一些较大尺寸的资源时，尽可能使用九宫格。

### 5. 会用作Sprite的UI元件尽量以最小尺寸切

对于大量的UI小元件，例如图标、按钮等，尽量让美术人员以最小尺寸切。所谓的最小尺寸，就是图片刚好包围下这个UI元件。因为在NGUI中，它会以UI元件在打包前的源文件尺寸作为控件的尺寸，这意味着，如果将一个明明只有100\*100像素的按钮图片放置在一张500\*500的UI图片中，除了按钮图片部分，其他地方都透明掉，这样制作成UI图集之后，NGUI中调用这个Sprite时，它的尺寸会被识别为500\*500。

### 6. 对于会用作Texture的UI图片尽量保持长宽都为2的N次方

在Unity中，如果图片的长宽不为2的N次方，那么将无法使用自带的压缩格式导致资源量变大，而且加载和处理效率都会慢上很多。所以，如果一个UI图片将会被当作Texture来使用，尽量让美术人员将图片做成长宽为2的N次方，如2048\*1024。

### 7. 和设计人员沟通，将UI元件尽量分类整理避免重复

一个游戏涉及了太多的界面，各种UI元件特别多。但是，其实有大量的UI元件是重复的，例如，很可能发现在很多界面中用的按钮其实都是同一个按钮。所以，客户端人员在开发UI时，需要和设计人员

沟通，建议他们尽量将UI元件分类整理一下，例如，游戏中一共会存在3种形式的按钮、两种形式的界面底框、5种形式的文字底框等。

### 6.5.3 程序如何保证UI资源的分辨率不失真

当UI图片导入到引擎中时，可能有时候会遇上美术人员在用PhotoShop设计制作时尺寸刚好，但是放到客户端中就匹配不上的情况。在这种情况下，首先确保一点：Unity 中 Game视图的分辨率设置是项目中统一的分辨率，美术人员也是按照这个分辨率作为画布标准来设计的UI。然后进行如下操作。

如果是2DUI，那么只需要单击控件的Snap即可，让图片还原为原尺寸大小，效果几乎可以做到和美术人员用PhotoShop做的一模一样的效果。

如果是3DUI，因为相机不是正交相机，所以，因为距离、透视等关系，控件生成之后单击 Snap，控件尺寸还原到原文件的大小，但是即使这样，在游戏视窗中它依然会比源文件看上去更大。这种情况下，会导致我们无法还原美术人员的设计图，我们在使用3DUI时需要将3DUI的UIRoot下面的UICamera 的Field Of View 的值设为75，则控件的视觉大小将会和源文件应该有的大小保持一致，可以几乎完全地还原美术人员的设计效果。

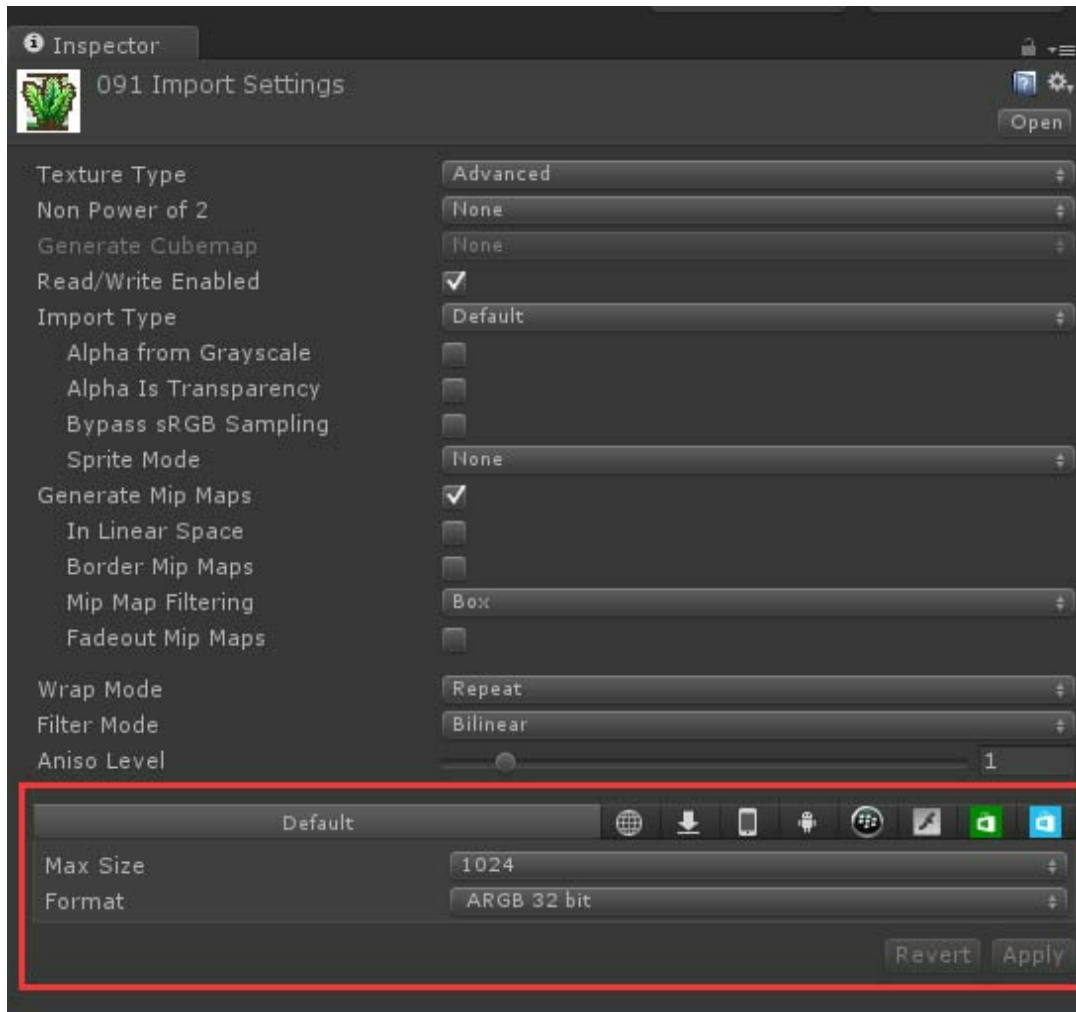
### 6.5.4 针对各大平台设置单独的尺寸和格式

在Unity中，跨平台时可以为每个图片设置不同平台下的资源和格式，比如一张1024\*1024的图片，可以让它在iOS平台下为1024\*1024，在安卓平台下就变为512\*512。

如图6.31所示，我们选中一个图片文件后，图 6.31中红框所示的部分即为各个平台的设置。

Default为默认的设置，向右依次是Web的设置、PC/Linux端的设置、iOS的设置、安卓的设置、黑莓的设置、Flash 的设置等。可以为图片设置其在不同平台下的尺寸和格式。如果没有设置，它将会在任何平台下都应用Default设置。

对于iPhone4手机：图片如果超过了2048尺寸，将无法显示（显示为一片黑色）。



▲ 图6.31

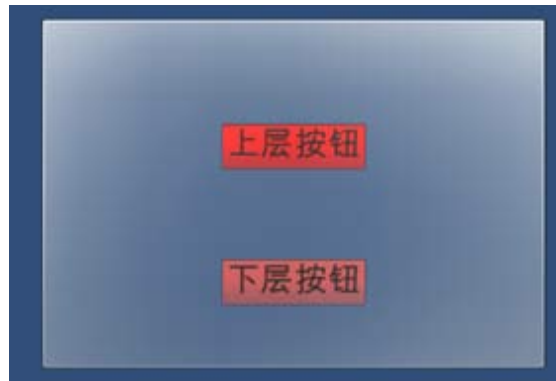
## 6.6 UI事件监听的击穿



### 6.6.1 什么是UI事件监听的击穿

在我们的游戏视图中，有两个UI界面叠在一起的时候，我们单击一个空白处，却触发了被覆盖在下层的UI界面中的单击事件，这就是我们的单击击穿了上层界面。

如图6.32所示，我们在界面上有一个“上层按钮”，然后整个界面的下层还有一个“下层按钮”，如果我们单击界面上下层按钮的位置，事件会直接击穿界面底板触发下层按钮的响应事件。



▲ 图6.32

再比如说，假设我们场景中放置了一个箱子，单击箱子会触发一个开箱事件，如果我们单击一个UI，恰好UI在视觉上将箱子覆盖了，那么它也许就会触发箱子的单击事件。这些都是我们不愿意看到的情况。需要一些方法来管理单击屏幕的事件监听。

### 6.6.2 如何避免和解决UI事件监听的击穿

通过NGUI的事件监听原理，我们知道它是通过BoxCollider来进行事件响应的。我们有好几种方法去应对事件击穿的情况。

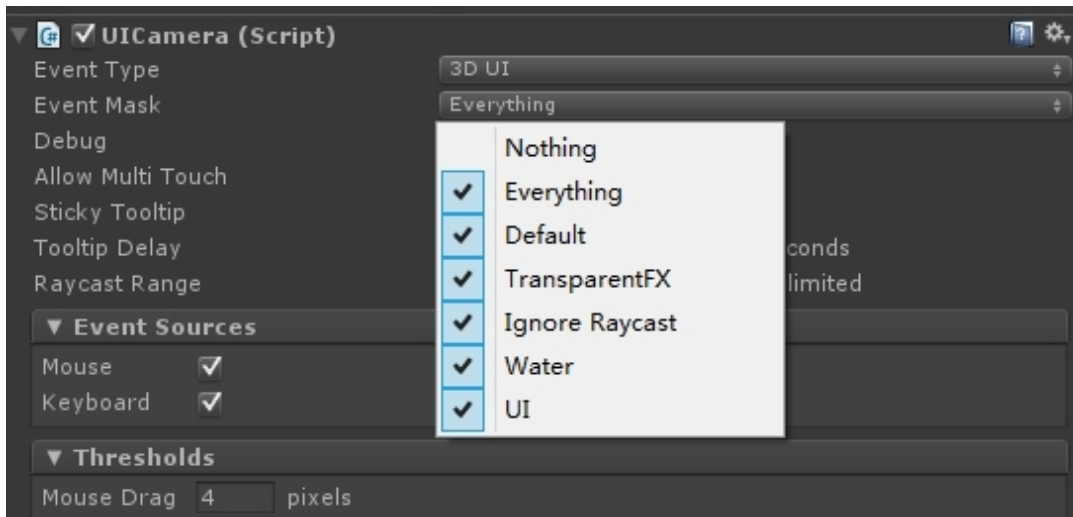
第一种方法：用一层BoxCollider覆盖，进行遮挡。

在图6.32中的例子中，如果在白色的界面底板上Attach一个BoxCollider，然后再单击下层按钮的位置，它的单击射线就会击中白

色的界面底板的 **BoxCollider**，从而避免触发下层按钮的单击事件。

第二种方法：使用**EventMask**。

我们知道Unity的Camera可以设置**CullingMask**来决定渲染的层，对于NGUI的事件监听核心组件**UICamera**来说，也有一个**EventMask**的设置。首先将下层物体和上层物体设为不同的层，然后在**UICamera**中设定这个摄像机只接收特定层的事件即可，如图6.33所示。



▲ 图6.33

### 6.6.3 事件监听遮挡的妙用

我们可以利用UI事件监听遮挡的功能实现一些小的妙用，这里举一个例子。在很多游戏中，我们都碰到了如下需求：假设玩家打开了一个界面，然后玩家如果单击了这个界面以外的任何区域，都将导致这个界面关闭。

对于这种小需求，我们就可以妙用事件监听，在界面的下面放置一个覆盖全屏的大型**BoxCollider**底板，然后给这个**BoxCollider**物体写一个单击事件，就是关闭打开的这个界面。这样当玩家单击到界面以外的任何区域时，它都会响应关闭界面的事件。

## 6.7 开发之前的思考——UI结构设计

### 6.7.1 什么是UI结构设计

我们在制作UI时，会涉及大量的UI控件，它们每一个控件都是一个独立的GameObject，往往一个场景内的UI内容极其复杂，做到后期的时候GameObject的结构更复杂。所以，我们需要提前有一套UI结构的设计，以此来方便对UI体系进行修改、添加和维护。

### 6.7.2 UI结构设计遵循的一些要点

1. 尽量不要让UI作为Camera的子物体。

在旧版本的NGUI中，制作2DUI时，默认UI是在Camera下面作为子物体创建。但是，在新版本的NGUI中，官方已经不默认这样做。因为UI和摄像机敏感的关系，尽量不要将UI作为摄像机的子物体，避免出现一些因为透视（3DUI）等问题导致的视觉Bug。

2. 尽量让Anchor组件所在的物体在最上层

因为屏幕自适应的适配工作是游戏避免不了的一个问题，Anchor组件所在的物体因为它具有定位自适应的重要功能，所以不宜太多，否则很乱不宜管理。最简单的方法就是尽量确保Anchor处于最上层。

3. 给UI分类，多做节点管理

比如一个场景中，有很多按钮菜单，每一个菜单点开之后都会出现一个相应的子界面，所以，可以用一个空物体（注意要和UI同层）作为菜单的总节点，然后用另一个空物体作为所有子界面的总节点。给每一种UI类型都做一个节点以此来管理，避免各种UI直接放在一起难以管理和维护。

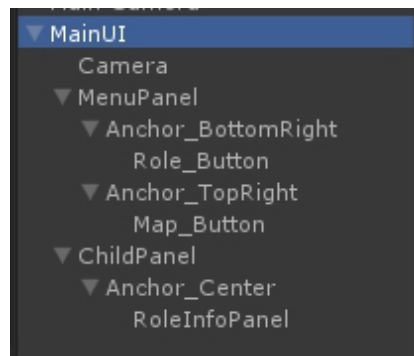
4. 重视深度管理并巧用Panel

我们知道界面在画面上的显示次序是根据深度来的，这个深度主要指Panel的深度和每一个控件的深度。其中Panel的深度优先，只要是深度更大的Panel，它下面的所有子UI物体，几乎都将显示在更上层。利用这一点可以轻松地放置一些UI之间的层次错乱问题，例如，主界面的按钮菜单和屏幕中央的子界面面板不能重叠，可以让所有的菜单按钮都处于一个深度更低的Panel之中，然后让子界面内容处于一个深度更高的Panel之中。

### 5. 重视命名

因为所有的控件本质上都是一个GameObject，NGUI的实现全是依赖一个又一个的组件来实现相应的控件功能。所以，这将导致我们的UI从名称上识别起来很困难，这个时候命名的作用就体现了出来。例如，一个Label物体，希望用这个Label来显示名称，可以给这个Label物体命名为：Name\_Label，这样我们一眼看过去就能知道它的类型（label）和用途（显示name）。

图6.34展示了一个比较简单的UI结构的范例。



▲ 图6.34

## 6.7.3 需要的时候，分场景以减轻内存负担

因为UI图集在使用时，会以一整个图集全部加载到内存当中。如果有10个系统，每一个系统的界面都是全屏界面，都有一张巨大的界

面独有的背景图，那么这10个系统的界面所用到的资源将会导致内存很大。

而Unity有一个机制是切换场景之后，会清除以前的内存，所以，在适当的情况下，可以考虑分场景来制作UI，以减轻内存的负担。相当于点开一个UI实则是进入了一个新的场景，看上去就像是全屏界面的效果一样，退出UI面板就相当于回到了之前的场景。

# 第7章 用代码深度控制UI

## 7.1 代码操作NGUI的原理

### 7.1.1 物体与组件的概念

Unity引擎的核心架构理念就是游戏物体（GameObject）与组件（Component）构成世界，物体是游戏世界中的核心单位，也几乎是惟一单位，灯光是一个物体、角色是一个物体、地形是一个物体，GameObject构成了整个游戏世界。

而组件，则是物体产生工作的核心单位，每一个物体都有一个或者多个组件，这些组件决定着这个物体拥有哪些功能，如最基础的Transform组件。

NGUI中，每一个UI控件，本质上都是一个GameObject，它工作的原理是这些GameObject上被赋予了NGUI的一些脚本组件。

所以，可以通过正常地操作物体一样，获取物体、获取它身上的组件类，然后修改其中的成员实现对NGUI的控制。

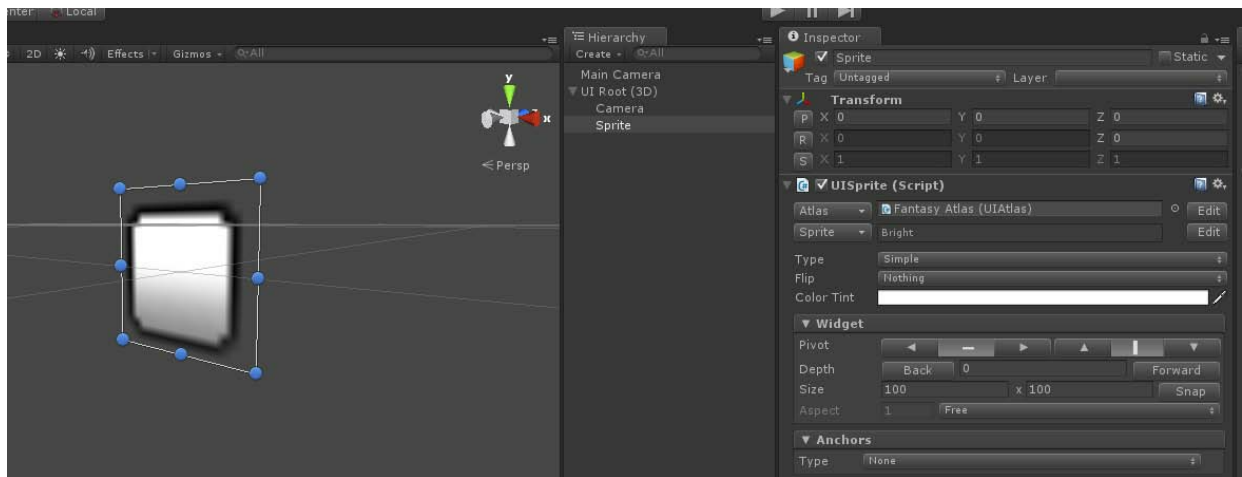
### 7.1.2 怎样用代码操作NGUI

我们来演示一个最简单的例子，用代码来修改一个Sprite控件的颜色，如图7.1所示。图7.1中我们在场景中创建了一套UI，并新增了一个UI控件，它目前是白色的。

首先我们先创建一个 C#脚本（JavaScript也可以，但是，因为C#在商业级项目开发中有更广泛的应用，所以，本书均使用C#作为讲解），命名为ChangeColor，然后双击打开：

```
using UnityEngine;
using System.Collections;

public class ChangeColor : MonoBehaviour {
    // Use this for initialization
    void Start () {
    }
    // Update is called once per frame
    void Update () {
    }
}
```



▲ 图7.1

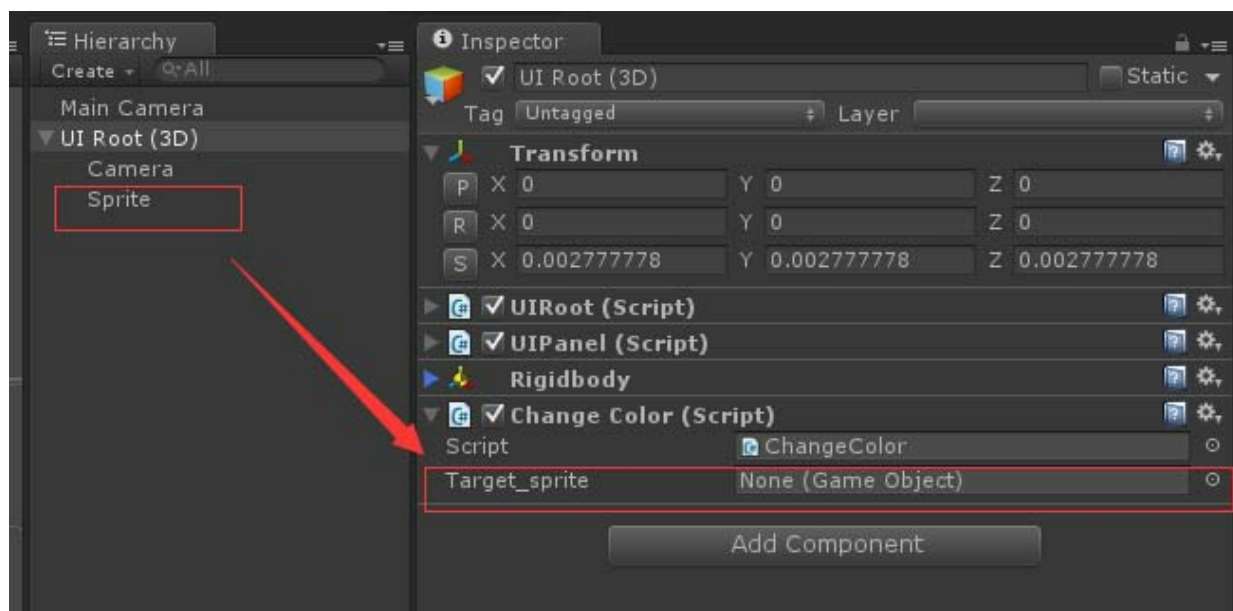
这是一个 Unity中创建的 C#脚本的默认状态，默认生成了一个和文件名同名的类、一个Start()函数、一个Update()函数。其中Start只会在第一帧Update运行前运行一次，而Update会在游戏中每一帧都运行一次，这些基础的Unity代码知识这里就不多赘述了。



首先，我们要在类中声明一个GameObject类型的共有变量，命名为：**target\_sprite**，我们将会使用这个变量作为要修改的 **Sprite** 的物体引用，然后获取这个 **Sprite** 物体身上被赋予的NGUI的**Sprite**组件，在**Start**中修改它的颜色为红色，最终代码如下：

```
using UnityEngine;
using System.Collections;
public class ChangeColor : MonoBehaviour {
    public GameObject target_sprite;
    // Use this for initialization
    void Start () {
        //获取目标物体的Sprite组件
        UISprite theSpriteComponent =
target_sprite.GetComponent<UISprite>();
        //修改Sprite组件的color成员变量
        theSpriteComponent.color = Color.red;
    }
    // Update is called once per frame
    void Update () {
    }
}
```

然后我们将这个脚本挂在任意一个物体上（最好挂在最上层物体，方便寻找和管理），然后将目标**Sprite**拖入到组件中的共有变量引用中，如图7.2所示。



▲ 图7.2

最后运行游戏，我们会发现，精灵图片从最开始的白色，变成了红色。

### 7.1.3 获取组件的几种方法

由上一小节例子可以看到，我们获取NGUI组件的方式是声明Public变量去引用物体，获得物体身上的组件，然后对组件进行修改。而在游戏开发中，经常会涉及大量的UI控件，如果全部都用Public变量去进行引用的话，会造成序列化过程耗时太长，并且极其难以维护。如图7.3所示的UI物体结构，控件量非常大，很难全部Public变量去保存引用。



▲ 图7.3

因为获取组件这个操作是NGUI的代码调用里最最核心的一步前提操作，下面来了解一下几种获取UI组件的方法。

### 1. 方法1：引用物体

声明Public物体变量，然后把物体拖上去保存该物体的引用，用Awake()或者Start()函数获取组件。

## 2. 方法2: 引用组件

声明一个Public的、明确的组件类型的变量（如声明一个public UIButton myBut），然后将物体拖上去即可保存该组件的引用（前提是这个GameObject上确实有该类型的组件）。值得注意的是，NGUI的组件一般都是以UI开头的，如UISprite、UIButton等。

## 3. 方法3: 寻找子物体

通过FindChild方法来一级一级地找到组件，例如上一小节中，将ChangeColor脚本挂在了UIRoot上，那么可以修改代码成如下来获取引用：

```
using UnityEngine;
using System.Collections;

public class ChangeColor : MonoBehaviour {
    public UISprite target_sprite;
    // Use this for initialization
    void Start () {
        //先找到目标物体上的组件
        target_sprite =
transform.FindChild("Sprite").GetComponent<UISprite>();
        //修改Sprite组件的color成员
        target_sprite.color = Color.red;
    }
    // Update is called once per frame
    void Update () {
    }
}
```

#### 4. 方法4: Find方法 (耗时, 不推荐)

使用GameObject.Find()方法来寻找一个特定的特殊UI物体, 但是, 这种方法将会遍历场景中的所有物体, 非常耗时, 不推荐使用。

#### 5. 方法5: FindWithTag (尽量不要使用)

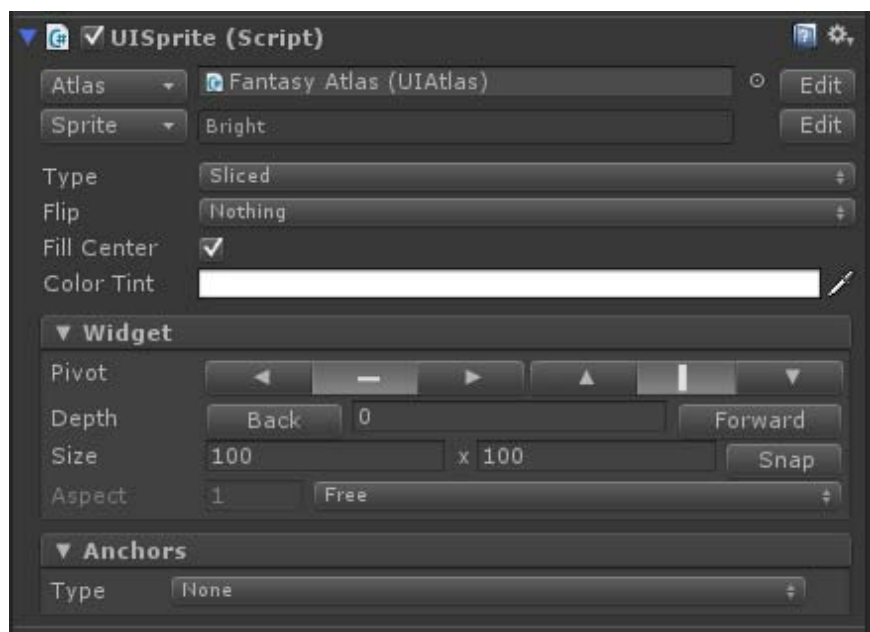
使用 GameObject.FindGameObjectWithTag 方法去寻找带有某种特殊标签的物体, 这种方法性能上比Find更快一点, 但是与本章前3个方法相比并没有优势, 并且会涉及标签Tag的管理工作, 不是很利于维护修改, 尽量少使用。

### 7.1.4 迅速判断可以修改的成员

当我们可以获取NGUI的组件时, 那么需要知道这个组件到底哪些成员是可以供我们修改的, 否则, 即使获取到了组件也没有用。而我们并不可能在需要时去详细阅读NGUI的源代码, 那样将会非常耗时间, 而且也失去了这个插件便捷开发者的意义了。所以, 需要学会判断哪些成员是可以修改的成员。

一般来说, NGUI的每一个组件, 都将会会有一个组件设置界面, 如图7.4所示的UISprite界面。

NGUI 的组件中, 提供大家可以修改的成员有两部分, 一部分是直接暴露在组件界面中的, 例如图7.4中的界面中的各个设置项分别对应着: atlas、spriteName等变量。也就是说: NGUI组件的设置界面中的几乎所有设置项都有一个相应的成员变量或者成员函数可以进行控制和修改。如果你有需求去修改某一个设置项, 而你又不知道它对应的是哪个变量, 那么可以直接试着输入这个设置项的名称, 一般来说IDE (VS和Mono均可以) 会自动提示出相应的变量和函数。



▲ 图7.4

而另一部分可以修改的成员没有暴露出来，需要翻看源代码才能知道。我们将在下面罗列出常用的组件中的可修改成员变量。

## 7.2 动态加载UI元素

### 7.2.1 为什么游戏中会用到动态加载UI元素

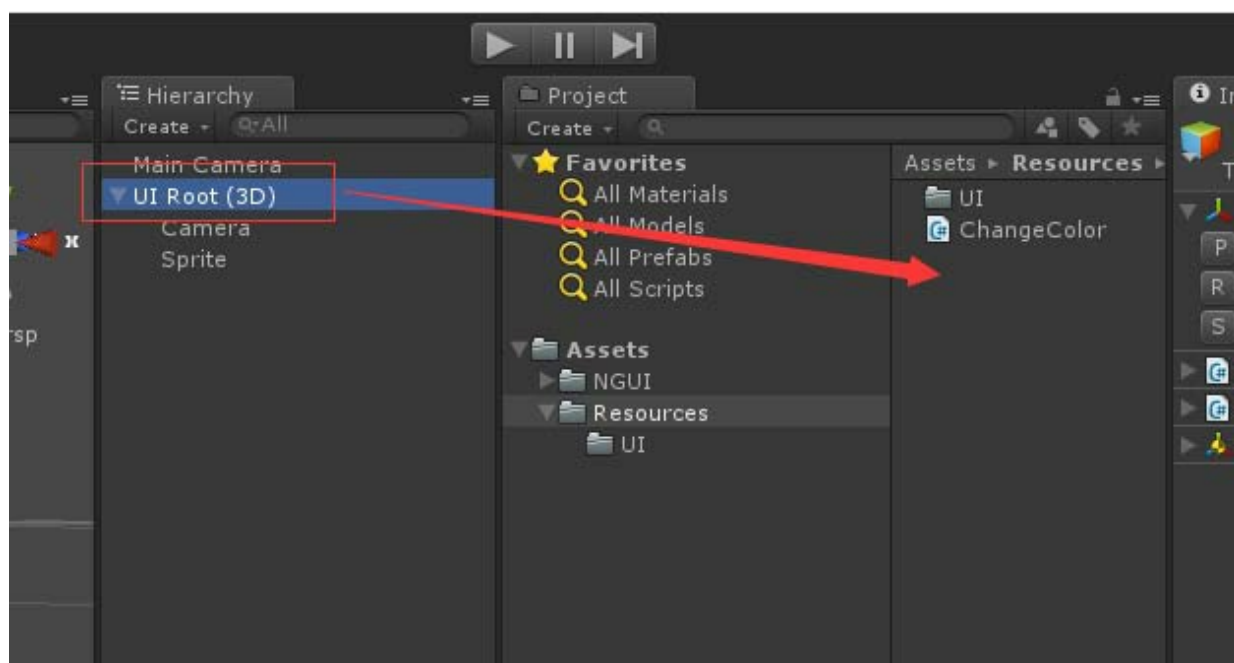
在游戏中，我们经常会用到动态加载UI元素的预设体，比如进入一个场景后，场景内的UI是动态加载的。又比如单击一个页面，会根据玩家的信息动态地生成一系列菜单，这个时候菜单UI因为会随着玩家的信息不同而改变，可能就会用到动态加载和摧毁UI。

为什么我们会用到动态加载UI？因为，如果我们预先就把所有的UI都在场景中做好，那么在加载这个场景时，这些所有的UI都将会在加载过程中全部加载进去并在初始状态时一次性全部部署好，如果UI

体系过于庞大，这样可能导致会有卡顿的体验。所以，避免不了会在一些特殊的地方使用到动态加载。

### 7.2.2 擅用UI元素的Prefab

直接将UI物体拖到Project视图中，则会自动生成一个Prefab，这就是这个UI的预设体，如图7.5所示。



▲ 图7.5

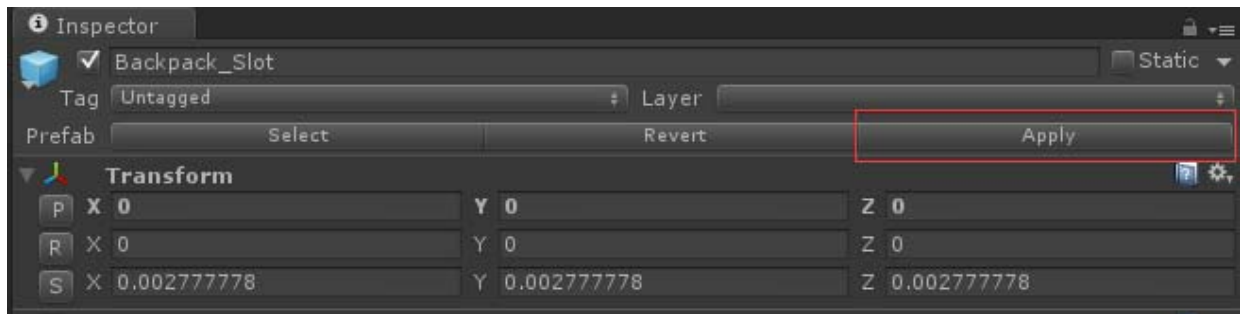
我们为什么要用 UI元素的预设体？首先，只有将UI元素事先保存为预设体，才能够允许在游戏运行过程中由代码去控制动态加载。

其次，游戏中的UI体系一般都非常庞大，而且，其中有非常多的重复性，所以，在游戏开发中要学会擅用UI元素的Prefab。例如，一个背包有100个格子，可以只做一个格子，然后将这个格子保存为一个Prefab，然后自动生成100个。

使用UI元素的Prefab还有一个非常好的地方在于：修改的时候，可以只修改一个就将所有的同类元素修改了。我们依然拿背包做例子，假设保存了一个背包单个格子的 Prefab，现在我们得到需求：需要给每



一个背包格子加上一个外框，这个时候如果我们一个格子一个格子地去加，将会非常浪费时间而且容易出错，我们就可以在Hierarchy窗口中对其中的一个格子的预设体进行修改，修改完了之后单击如图7.6所示的Apply按钮即可自动让所有用到这个预设体的UI都更新到最新的状态。



▲ 图7.6

### 7.2.3 将一个物体设置为另一个物体的子物体—— NGUITools.AddChild()方法

NGUITools.AddChild()是NGUI提供的工具类中最常用的一个方法，它可以直接将一个物体设置为另一个物体的子物体，并且这个子物体的位置直接就在父物体所在的位置。

NGUITools.AddChild函数可以传入两个参数：父物体和子物体，例如，如下所示的脚本展示了动态加载NGUI的元素的使用方法，该脚本动态加载了一个UI元素到Root下：

```
using UnityEngine;
using System.Collections;
public class LoadPrefab : MonoBehaviour {
    //声明UIRoot这个物体的引用，待会儿将会在这个物体下生成子物体
    public GameObject uiRoot;
```

```

//声明要加载的子物体预设的名称
string prefabName = "Template";
void Start () {
    if (uiRoot!=null)
    {
        //根据路径将预设加载进内存作为一个GameObject存在
        GameObject go = Resources.Load("UI/" + prefabName) as
GameObject;
        //使用NGUITools.AddChild方法挂子物体
        GameObject newObj = NGUITools.AddChild(uiRoot, go);
        //可以将新物体的名称打印出来
        Debug.Log("新生成了一个子物体名叫: " + newObj.name);
    }
}
void Update () {
}
}

```

#### [7.2.4 NGUITools.AddChild\(\)和Instantiate的区别](#)

我们知道Unity中本身有一个实例化预设的方法：Instantiate。那么就了解一下这两种方法。其实这两种方法本质上是完全一样的，只不过是NGUI.AddChild方法进行了包装，以便更快捷地增加子物体而已。

NGUITools.AddChild(parent,child)方法主要用于快速给父物体增加一个子物体，如果要使用Instantiate方法来实现，则需要以下代码：

```
GameObject newObj2 = Instantiate(go, uiRoot.transform.position,  
uiRoot.transform.rotation)as GameObject;
```

```
newObj2.transform.parent = uiRoot.transform;
```

其实这两种方法本质上完全一样，NGUITools.AddChild 方法在源代码中也是通过Instantiate 去实现的。所以，在开发中不必为这两种方法纠结，根据个人喜好和需求灵活决定使用哪种方法即可。

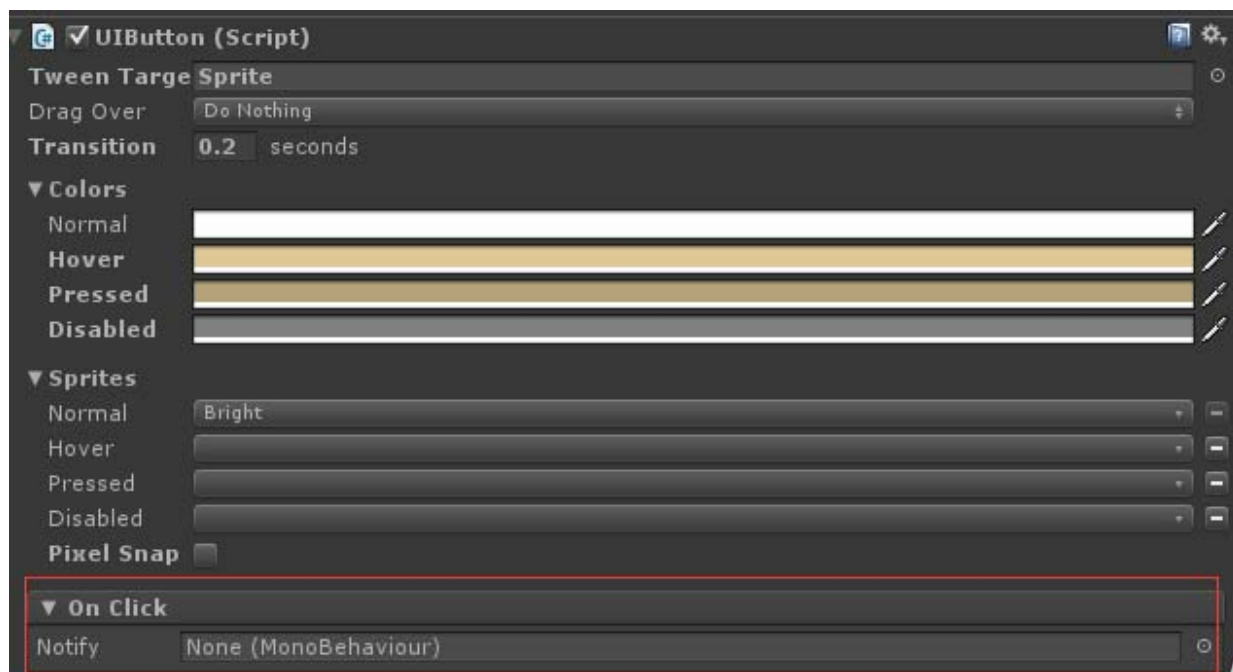
## 7.3 擅用EventDelegate事件委托

### 7.3.1 什么是EventDelegate事件委托

EventDelegate事件委托是NGUI 3.0以后统一的底层传递消息、监听事件的机制，在NGUI 3.0 版本以前NGUI用的是SendMessage 方法，但是效率太低。NGUI 3.0 版本以后全部改为了EventDelegate机制来进行事件的监听，效率提高了100倍。

EventDelegate本质上和C#的Delegate没有任何区别，它将会负责NGUI中所有的事件监听、回调等。例如，单击Button按钮触发一个事件，那么Button组件中就会有OnClick的回调。

对于熟悉 C#中委托的读者可能会更轻松地理解 EventDelegate，不熟悉的读者，可以将EventDelegate理解为一个特殊类型的变量，它可以赋值在任何一个带有如图7.7所示的Notify界面中。



▲ 图7.7

### 7.3.2 事件委托的用法

使用事件委托，可以将一个函数整体当作一个变量，来赋值给各个组件的回调模块，也就是图7.7中红框所示的Notify模块。NGUI组件的Notify模块是一个事件回调的List，它可以支持多个回调事件，例如一个按钮的OnClick的Notify中，可以挂载无数个事件。

以下代码示范了如何将一个函数，由代码动态赋值给Button的单击回调事件组，让Button单击后能调用这个函数：

```
using UnityEngine;
using System.Collections;

public class ButtonClick : MonoBehaviour {
    //声明UIButton这个组件的引用，待会儿将会为这个组件的单击
    事件赋值
    public UIButton myButton;
    void Start () {
```

```

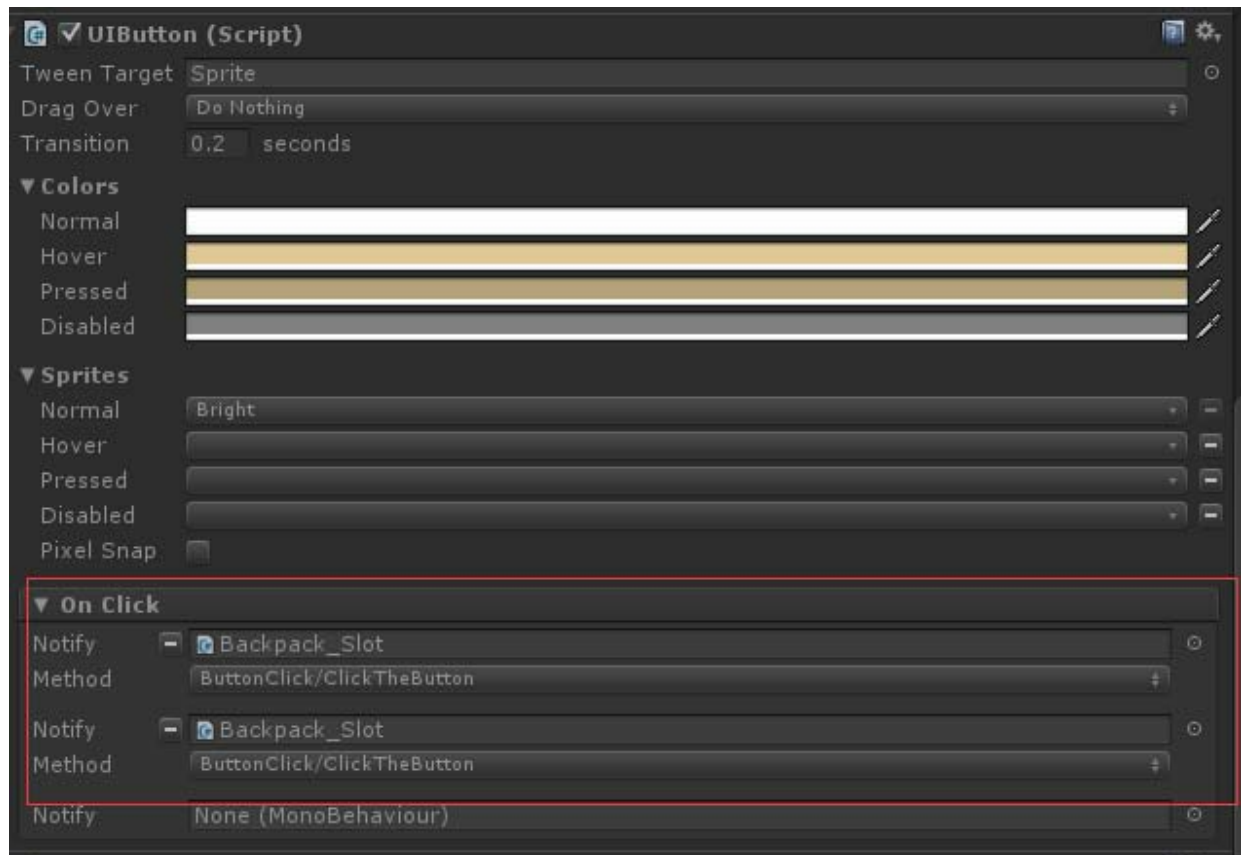
        if (myButton != null)
        {
            //首先将本脚本中的ClickTheButton()方法变成一个
            EventDelegate类型的事件委托
            EventDelegate theED = new EventDelegate(this,
            "ClickTheButton");
            //方法1:  EventDelegate.Add (组件的Notify回调组名称 , 一个
            EventDelegate类型的事件)
            EventDelegate.Add(myButton.onClick, theED);
            //方法2:  因为Notify回调本身就是一个事件组, 所以直接Add
            一个EventDelegate类型的事件
            myButton.onClick.Add(theED);
        }
    }
    void Update () {
    }
    //要作为EventDelegate类型的单击事件, 一定要是Public的
    public void ClickTheButton()
    {
        Debug.Log("我单击了这个按钮! ");
    }
}

```

我们将这个脚本挂到一个物体上, 将一个带有BoxCollider和UIButton组件的物体拖到这个脚本组件的 myButton 引用中, 然后运行, 单击这个按钮, 将会看到控制台打印出一句话: “我单击了这个按钮!”, 证明我们通过代码动态地给UIButton赋予了一个单击事件。并且在运行状态下, 我们选中这个Button物体, 可以看到它的UIButton组

件的单击回调部分如图7.8所示，再次证明我们动态赋值单击事件成功（必须运行状态下赋值代码执行了之后才看得到）。

注：图7.8中我们看到OnClick在运行后被赋予了两个相同的单击事件，是因为我们的脚本中为了展示两种赋值的方法（上文代码中的方法1 和方法 2），所以被赋值了两遍，实际开发过程中要避免这一点。



▲ 图7.8

### 7.3.3 哪些地方可以使用事件委托

NGUI中，所有带有Notify回调模块的组件，都可以使用事件委托，比如Tween动画组件的OnFinished、按钮组件的OnClick、Toggle开关组件的OnValueChanged等。这些组件的事件回调全部都是EventDelegate类型的，使用方法和上一小节中按钮的单击事件赋值一样。

## 7.4 巧用EventTrigger组件

### 7.4.1 什么是EventTrigger组件

EventTrigger组件，是新版NGUI中提供的一种统一的监听各种事件的组件。增加方法为，依次选择

AddComponent → NGUI → Interaction → EventTrigger。它可以监听单击、按下、选中、拖曳等各种事件，组件界面如图7.9所示。

其中。

OnHoverOver: 鼠标光标移动到目标上之后触发。

OnHoverOut: 鼠标光标从目标上移开之后触发。

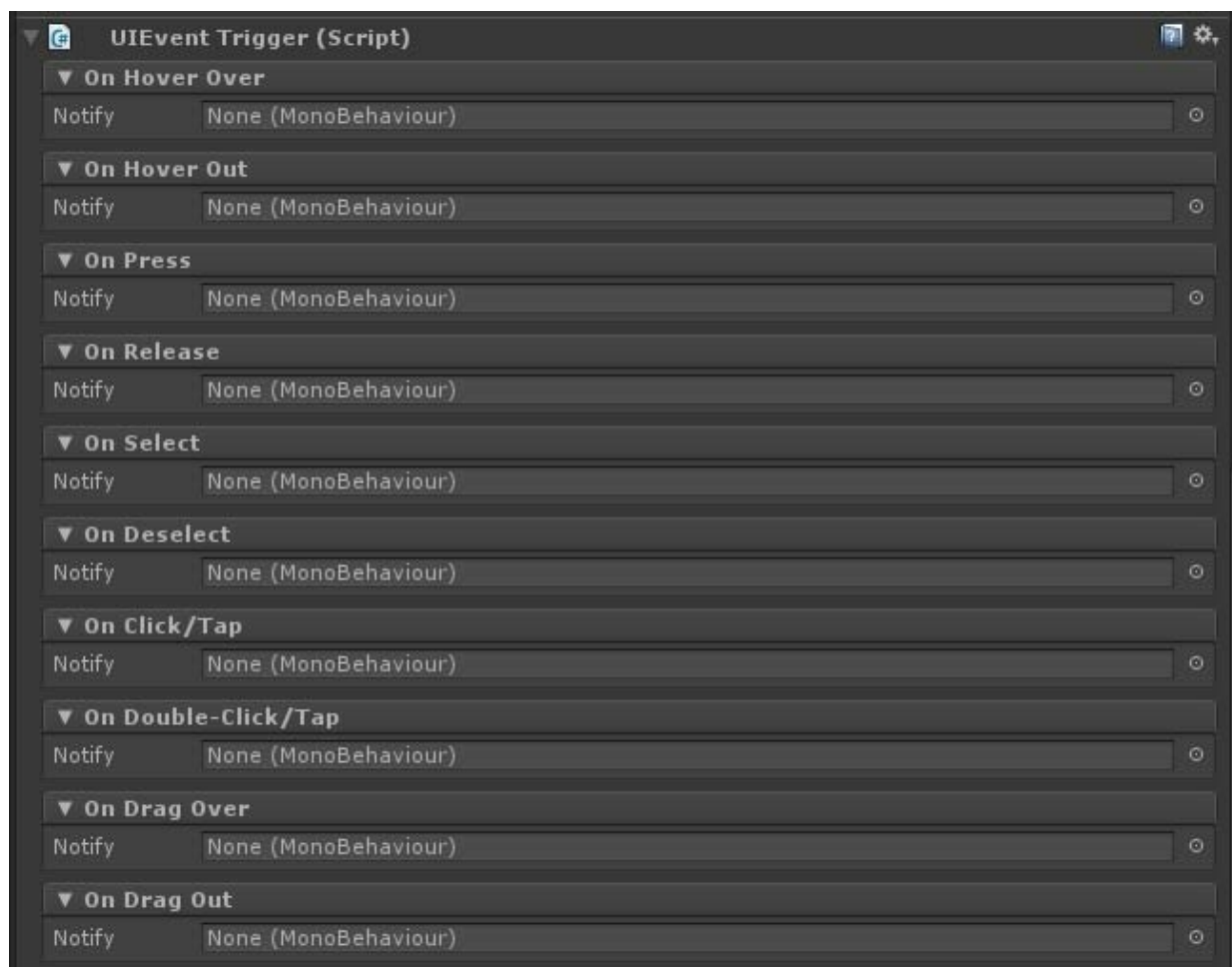
OnPress: 按下的时候触发一次，松开的时候再触发一次。

OnDragOver: 拖动停住时触发。

OnDragOut: 拖动松手后触发。

以上为最常用的效果，其他事件效果欢迎大家自己去尝试，这里就不多赘述。





▲ 图7.9

## 7.4.2 EventTrigger用法

EventTrigger本质上就是一个事件监听的集合，每一种事件都有一个Notify回调事件组，我们可以像为UIButton的单击事件赋值一样来为它赋值，比如可以拖动物体到这个Notify设置项中，会出现这个物体带有的所有脚本组件中的所有的公共函数（只有Public公共函数才会在这里显示，非Public是无法外部访问调用的），然后我们在这些出现的公共函数中选择一个我们想要在该事件触发时执行的函数，就完成了这个事件触发的赋值，具体代码如下：

```
void OnPress()
```

```
{  
    Debug.Log("Press");  
}  
void OnDragOut()  
{  
    Debug.Log("DragOut");  
}
```

## 7.5 常用组件的功能调用

### 7.5.1 UILabel

首先，我们假定获取了一个UILabel的实例label:

```
UILabel label= GetComponent<UILabel>();
```

- 显示/关闭文字显示:

```
label.enabled = false/true;
```

```
label.gameObject.SetActive(false/true);
```

第一种方法是关闭组件，第二种方法是直接隐藏物体。

- 赋值文本内容:

```
label.text = "这里是给文本赋予内容"
```

- 改变字号大小:

```
label.fontsize = 100;
```

这里需要注意的是，Label的区域限制类型，如果是ShrinkContent则字体被始终自动缩放以限制在控件范围内。

- 整体改变文本颜色:

```
label.color = new Color(1,1,1,1);
```

```
label.color = Color.red;
```

在改变颜色时，如果使用 `new Color` 给颜色赋值，要注意 `new Color` 需要的参数依次是RGBA值，其中最后的A值可以没有。对于RGBA值，需要将其从0~255转换为0~1的一个小数，转换方式为颜色的色值除以 255。另外，`Color` 提供了一些静态的色值供直接调用，如红色`red`、黄色`yellow`、绿色`green`等。

- 使文本某些文字不受颜色影响:

```
label.text = "不受影响的文字内容是: [c]这里的文字";
```

在这里我们在文本中插入了“关键字”`[c]`，所谓的关键字，就是指在文本中一旦出现，它就会被当成一种指令不会被显示在文本中，而会影响其后面的文字。在这里的例句中，关键字`[c]`后面的文字都将不受整体颜色的影响而保持白色。

- 局部改变某些字的颜色。

如果要改变`Label`文本中的字体颜色，可以插入十六进制的颜色值实现，例如：

```
label.text="我将会改变颜色为[FF0000]红色";
```

在这行代码中，`[FF0000]`会作为关键字不会在文本显示出来，它会将它之后的文本内容全部变成红色。

- 修改深度:

```
label.depth = 10;
```

## 7.5.2 UISprite

首先，我们假定获取了一个`UISprite`的实例`sprite`:

```
UISprite sprite = GetComponent< UISprite >();
```

- 显示/隐藏图集:

```
sprite.enabled = false/true;
```

```
sprite.gameobject.SetActive(false/true);
```

第一种方法是关闭组件，第二种方法是直接隐藏物体。

- 修改精灵图片：

```
sprite.spriteName = "newName";
```

修改图片的名称后，会自动调用当前图集下的该名称图片。如果当前图集下不存在该名称的精灵，则将不会显示任何图像内容。

- 修改颜色和深度。

和修改字体颜色深度一样。

### 7.5.3 UITexture

首先，我们假定获取了一个UITexture的实例texture：

```
UITexture texture = GetComponent< UITexture >();
```

- 获取并修改图像内容：

```
texture.mainTexture = Resources.Load("这里是新图片的路径") as Texture;
```

- 修改尺寸：

```
texture.SetDimensions(200,200);
```

这是一个所有控件（label和sprite）都可以使用的方法，需要传入两个参数：宽度和高度。

- 修改颜色和深度。

和文字的颜色深度修改方法一样。

### 7.5.4 UIButton

首先，我们假定获取了一个UIButton的实例button：

```
UIButton button = GetComponent< UIButton>();
```

既然是一个按钮控件，那必须带有BoxCollider来接收操作输入：

```
BoxCollider boxcollider = GetComponent<BoxCollider>();
```

- 让按钮禁用:

```
boxcollider.enabled = false;
```

特别需要注意的是，按钮的禁用是非常特殊的一种情况，因为，即使没有UIButton组件，也能通过OnClick等事件监听方式（如EventTrigger中那些）实现按键效果，所以，要禁用按钮，我们应该禁用的是接收事件的根本组件：BoxCollider。

也只有当BoxCollider被禁用了，UIButton才会变成Disabled的颜色。

- 给按钮赋予事件。

前文讲解EventDelegate已经讲过，不再赘述。

### 7.5.5 UIGrid

首先，我们假定获取了一个UIGrid的实例grid:

```
UIGrid grid = GetComponent< UIGrid>();
```

- 获取并改变网格的间距:

```
grid.cellHeight = 500;
```

```
grid.cellWidth = 500;
```

- 使网格立即重新排列:

方法1: grid.repositionNow = true;

方法2: grid.Reposition();

### 7.5.6 UISlider

首先，我们假定获取了一个UISlider的实例slider:

```
UISlider slider = GetComponent< UISlider>();
```

- 获取并更改进度条的进度值:

```
slider.value = 1.23f;
```

这里进度值的动态赋值应该是一个0~1的浮点数，因为进度的范围就是0%~100%。当对它进行实时赋值时，就是常见的血量条等。例如：

```
slider.value = currentHP/maxHP;
```

- 获取并更改进度条的透明度值：

```
slider.alpha = 1.35f;
```

- 获取并更改进度条每一步变动的步幅值：

```
slider.numberOfSteps = 10;
```

这里赋值可以设置进度条一共有几个点，也就是进度条将会一段一段地变化。例如，填入5，则进度条只会存在0、0.25、0.5、0.75、1 5 种情况。

- 进度条的进度值变动的事件回调：

```
slider.onChange.Add( new EventDelegate(this,"A") );
```

这里赋值的方法，与前面讲解EventDelegate一样。

- 可拖动进度条拖动结束后的事件回调：

```
slider.onDragFinished.Add( new EventDelegate(this,"A") );
```

### 7.5.7 UIToggle

首先，我们假定获取了一个UIToggle的实例toggle：

```
UIToggle toggle = GetComponent< UIToggle>();
```

- 获取并修改它的选中状态：

```
toggle.value = false/true;
```

- 获取并修改它所属的开关组：

```
toggle.group = 10;
```

选中状态改变后的事件回调：

```
toggle.onChange.Add( new EventDelegate(this,"A") );
```

### 7.5.8 UIInput

首先，我们假定获取了一个UIInput的实例input:

```
UIInput input = GetComponent< UIInput>();
```

- 获取并修改用户输入的内容的值:

```
input.value = "这里是输入框的值";
```

这是Input最常用的一个变量，因为经常需要获取玩家输入的内容，这个变量是一个string类型的。

- 改变文本颜色:

```
input.activeTextColor = new Color(1,1,1,1);
```

这个功能在登录、注册等功能上很常见，比如当用户输入的文本不符合某种要求时，就让文本变红，如账号错误。

- 提交文本和改变的事件回调:

```
input.onSubmit.Add( new EventDelegate(this,"A") );
```

```
input.onChange.Add( new EventDelegate(this,"A") );
```

### 7.5.9 UIPanel

首先，我们假定获取了一个UIPanel的实例panel:

```
UIPanel panel = GetComponent< UIPanel>();
```

- 改变Panel的透明度:

```
panel.alpha = 0.5f;
```

- 改变Panel的深度:

```
panel.depth = 100;
```

- 改变Panel的渲染次序:

```
panel.renderQueue = UIPanel.RenderQueue.StartAt;
```



```
panel.startingRenderQueue = 3000;
```

这个功能会和 **Depth** 有一些冲突，因为它是最决定渲染次序的。当我们使用粒子系统在UI界面上一起显示时，粒子系统的**RenderQueue**一般为3000，如果Panel的**RenderQueue**高于3000，则永远会遮挡粒子，反之则粒子会在Panel前面。

- 重新渲染画面：

```
panel.RebuildAllDrawCalls();
```

这是重新建立所有的**DrawCall**，我们知道**DrawCall**是CPU发送给GPU的绘图指令，可见这是一个非常霸道的功能，将会有更长的延迟消耗，所以，一般情况不要使用。它一般用在NGUI某些层级改变了之后，没有及时刷新的情况下。

例如，我们在游戏中，假设有**Panel A** 深度为1、**Panel B** 深度为2。我们动态地将一个控件由**Panel A** 下面移到**Panel B** 下面，我们有时候会发现这个控件并没有立即因为**Panel B**的高深度而立即显示在 **Panel A** 的上层，这个时候我们就需要手动刷新显示，就可以使用这个 **RebuildAllDrawCall()**方法。

### **7.5.10 UICamera**

**UICamera**组件一般会极少去调用它，下面来了解几个可能用得上的静态方法或者变量。

**UICamera.IsOverUI**，返回一个bool值，我们单击控件之类的操作在NGUI底层都是依赖射线实现的，这个方法是判断最后一个射线是否击打在UI上。

**UICamera.IsPressed(Gameobject go)**，这个方法是检验当前情况下，是否按住了某一个物体，比如传入这个物体作为方法的参数。

另外，我们可以像操控其他控件一样，去获取UICamera的实例组件，然后修改它的组件面板中的各项值，方法和操控其他组件一模一样。因为对UICamera的操作一般很少见，一般也就偶尔改一下Depth之类的，所以，这里就不多描述了，大家可以按照操控其他组件的方式自主进行尝试。

## 7.6 动画的控制

### 7.6.1 为什么要把动画单独提取出来

在这里，我们将 NGUI的动画（主要指Tween动画）单独讲解，主要是因为动画它有一个最大的特点：它不停地进行插值改变。例如，我们执行一个位置从（0,0,0）变换到（0,10,0）的 Tween 动画，当执行到一半的时候也就是移动到（0,5,0）的位置时，我们关闭动画组件，会发现物体停在了（0,5,0）的位置。

正因为动画的这种实时改变的特性，所以，经常会碰到以下很多问题。

- （1）动画播放完了，再播放第二遍就没有效果了。
- （2）动画播放到一半，再重新播放它还是会从一半的地方开始。
- （3）很多其他的问题.....

NGUI的动画体系，在游戏的运用中，除了一些基本的、简单的动画可以依靠组件激活时的自动播放去执行以外（如Pingpong和Loop动画），很多时候都需要代码去手动控制动画。

### 7.6.2 控制Tween动画

获取Tween动画组件最简单的方法就是直接获取指定类型的组件，比如对于TweenPosition组件，我们可以通过GetComponent<TweenPosition>()来获得这个位移动画的组件。

Tween动画都继承自UITweener类，所以，如果一个物体身上有很多种Tween动画，而我们要一起控制它们，可以直接获取物体身上的UITweener。比如一个物体身上有3个Tween动画，一个位置动画、一个放缩动画、一个变色动画，我们可以通过 GetComponent<UITweener>()获得一个数组，这个数组内容就是物体身上所有的Tween动画。这里一定要注意GetComponent（获取多个组件）和GetComponent（获取单个组件）的区别。

需要注意的是，如果获取具体类型的组件，可以修改所有值，如果获取的是UITweener，因为这是一个基类，所以只能修改所有动画都共有的那些变量。

- 激活关闭Tween动画组件：

tween.enabled = true/false;

这一个方法在针对Loop和Pingpong动画时极其有用，因为Loop和Pingpong动画都是没有结束的，它们属于激活就播放、禁用就暂停。需要注意的是，禁用后物体的状态将会停在动画被禁用的那一刹那，如果禁用后希望动画复原需要手动用代码进行复原。

- 播放Tween动画。

对于一次性播放的动画，也就是Once类型的动画，播放一次后它将不会再播放第二次，这个时候如果我们需要用代码手动去控制它播放，则可以使用：

正向播放：tween.Play(true);或者tween.PlayForward();

反向播放：tween.Play(false);或者tween.PlayReverse();

如果有多个同类组件，例如有3个TweenPosition组件，我们希望在一定的场合下只控制其中一个去播放，可以给这3个组件分别设为不同

的Group，例如分别设为1、2、3，然后再进行判断播放，具体代码如下：

```
UITweener tweeners = GetComponents<UITweener>();  
for (int i = 0; i < tweeners.Length; i++ )  
{  
    if (tweeners[i].group==1)  
    {  
        tweeners[i].Play(true);  
        break; //找到需要播放的动画后，就跳出循环  
    }  
}
```

#### ●复原Tween动画。

动画是不会自动复原的，所以，播放时如果碰到没有复原导致的现象时，可以手动通过代码进行复原：

```
UITweener tween = GetComponent<UITweener>();  
tween.ResetToBeginning();
```

这种方法会使动画回到原点，这个原点并不是指的是起点，如果动画是正向播放，这个原点就是起点，如果动画当时是反向播放的，这个原点就是终点。

#### ●改变Tween动画变量。

首先我们可以轻易地改变UITweener部分的变量，例如修改动画持续时间：`tween.duration = 1.0f`。

例如修改动画的所属组：`tween.group = 3`。

如果要修改非Tween部分的变量，就需要获取具体类型的组件，例如，假设要修改一个位移动画的初始值和结束值，这个时候获取Tween就没有用了，因为UITweener是一个基类，所以，这种情况下需要获取这个位移动画：

```
TweenPosition tween = GetComponent<TweenPosition>();
```

```
tween.from = new Vector3(1,1,1);
```

```
tween.to = new Vector3(2,2,2);
```

●Tween动画播放完毕的事件回调:

```
tween.onFinished.Add( new EventDelegate (this,"A") );
```

这里需要注意的是，Loop和Pingpong动画是没有结束的。

### 7.6.3 关于PlayTween和PlayAnimation

关于PlayTween和PlayAnimation最重要的就是两个用法：播放和复原。对于播放，如果希望手动控制动画进行播放，那么PlayTween和PlayAnimation最好是Forward方向的，然后通过Play(true/false)来进行正向/反向播放。

我们可以在播放前通过 `playtween.resetIfDisabled = true` 来使动画在播完禁用后自动复原。

我们可以使用onFinished来对PlayTween和PlayAnimation进行结束事件的回调，这样就可以避免了去对具体的Tween动画进行结束事件设置，更易于管理和维护。

关于动画的一些更多的成员变量和函数，大家可以多进行尝试。

## 第8章 实用案例演示

在本章中，我们将会进行一系列的案例实战讲解，包含目前主流游戏中的各种主流模块。由于篇幅有限，我们只会挑选最常用的游戏UI模块来讲解制作，其中将会包括：

- 模块需求分析
- UI制作讲解
- 使用逻辑讲解
- 源码讲解

需要说明的是，本章讲解的案例，需要Unity 4.0 及更高的版本，需要NGUI 3.6.0 及更高的版本。本章使用的是Unity 4.5.4 和NGUI 3.7.2。为了让大家都能够更方便地学习案例，在本章所有的案例中，我们都将使用NGUI自带的资源，不会使用特殊资源。

NGUI大多数的知识点都已经在前文当中包含了，所以本章只挑选了几个UI模块来讲解。如果读者对NGUI的一些模块有疑问，建议详细回顾前面章节中与该模块相关的内容，或者直接参看第9章一些常见问题的解答。

如果需要更多的案例，欢迎进入Project面板：  
Assets/NGUI/Examples/Scenes 中观看官方提供的一系列案例。

### 8.1 角色头像状态栏制作

#### 8.1.1 示意图和需求分析

在本章中，我们将会制作一个如图8.1所示的角色头像状态栏界面。这个界面在RPG游戏中几乎是必备界面。

下面我们来了解一下这个界面的功能需求：

- 显示角色的头像；
- 显示角色的名称；
- 实时更新角色的血量百分比，精确到小数点后两位；
- 实时更新角色的魔法量百分比，精确到小数点后两位；
- 界面开发分辨率标准为1920×1080；
- 这个界面模块始终处于屏幕左上角；
- 角色名称最多4个字。



▲图8.1

### 8.1.2 设计并制作UI

我们将会按照以下步骤来制作UI。

#### 1. 创建UI

我们在场景中创建一个3DUI，将UIRoot命名为MainUI。

#### 2. 设置分辨率



因为开发分辨率标准为1920×1080，所以，将Game的视窗大小调整为1920×1080，并且将UIRoot的缩放模式（Scaling Style）改为“缩放”，并设置Height为1080。

### 3. 设置锚点

由需求知道这个界面模块始终处于左上角，所以，在UIRoot下创建一个Anchor，创建方法为Unity顶部，依次选择NGUI菜单→Creat→Anchor，并将Anchor设置为TopLeft。为了方便管理，将Anchor物体命名改为Anchor\_TopLeft。

### 4. 制作头像

我们为了管理这个模块，先在Anchor下创建一个Panel，命名为RoleStateUI，意为角色状态界面。然后在Panel下创建一个Sprite，Atlas选用NGUI自带的FantasyAtlas，Sprite选择Glow，这就是我们要拿来当作头像图标底的精灵，为了美观，我们将这个Sprite设置为Sliced模式，并命名为Head。

然后我们移动它到左上角最合适的位置上，在它下面创建一个子物体Sprite，命名为HeadIcon，然后Sprite图片选用OrcArmor-Bracers。我们就用这个图片来当作角色的头像图标。

### 5. 制作血条

我们在RoleStateUI下面再创建一个Sprite，命名为HP\_Slider，意为血条的意思，图片选用图集集中的Glow，设为Sliced模式，然后调整尺寸到一个我们需要的血条大小，它将会是角色血条的底子。然后在这个HP\_Slider上增加一个组件：UISlider，增加方式为，点中这个物体，在Inspector面板中依次单击AddComponent→NGUI→Interaction→NGUI Slider。

在这个血条底子HP\_Slider下创建一个子物体Sprite，命名为Foreground，它将作为表示血条进度的条子，Sprite图片选用图集集中的Glow，设置好尺寸大小后，将它的颜色改为红色，表示这是一个血

条。并将这个 **Foreground** 物体拖动到 **HP\_Slider** 物体的 **UISlider** 组件中的**Foreground**设置项中，将它和进度条组件关联起来。

在这个血条底子**HP\_Slider**下再创建一个子物体**Label**，命名为**PercentLabel**，意为百分比文本，调整文本的大小和位置，为了让文字看上去更清晰，我们可以去掉**Gradient**的勾让它纯白色显示。

#### 6. 制作魔法条

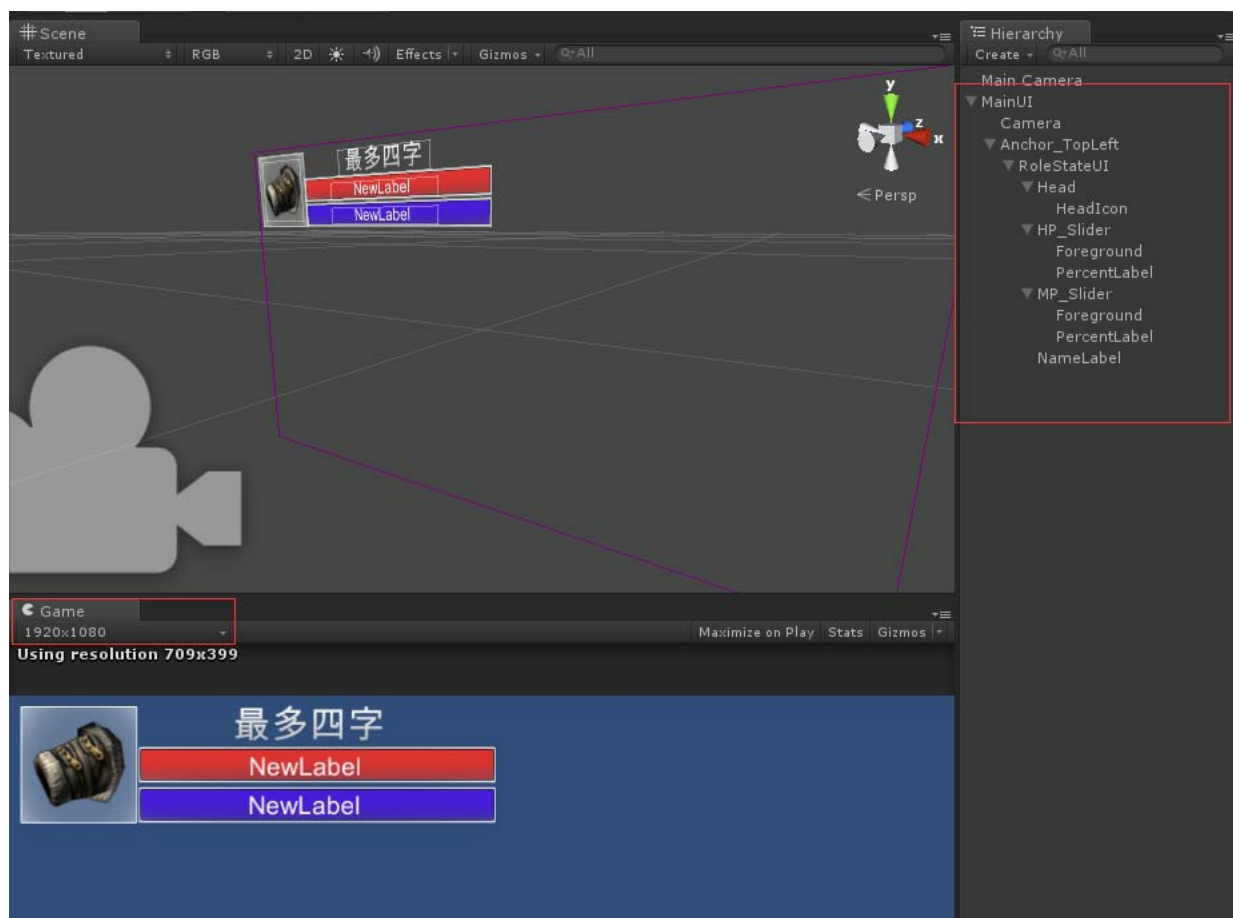
将刚制作好的血条复制一份（直接使用**Ctrl+C**、**Ctrl+V**复制即可），然后调整它的位置到血条下方，将它的物体名称重新命名为：**MP\_Slider**。

#### 7. 制作角色名文本

在**RoleStateUI**下创建一个**Label**，命名为**NameLabel**，调整这个文本物体到血条上方，调整好尺寸大小，并去掉**Gradient**的勾让它没有渐变纯白显示。

因为角色名称最多4个字，所以，我们需要提前输入4个字上去，看看**Label**控件的尺寸是否合适。

制作好后的UI结构图就如图8.2所示。



▲ 图8.2

这个UI结构目前来看，是一个比较合理并且清晰的UI结构，我们来梳理检查一下UI有没有少关键点。

- (1) 分辨率相关的设置（UIRoot）。
- (2) 锚点设置。
- (3) 血条上的Slider组件是否关联了Foreground。
- (4) 蓝条上的Slider组件是否关联了Foreground。
- (5) 各个Label的大小和最大文字数量是否已经测试好效果了。

注意：因为大多数时候NGUI会自动给新增加的控件的Depth增加1，所以，这里我们没有去检查Depth，但是在实际开发中，大家要养成检查Depth的习惯。

### 8.1.3 设计并编写代码

声明：本章节所涉及的代码部分，均只为它所属的案例服务，主要目的是给读者讲清楚一个简易的UI模块功能的完整开发的思路，并不代表实际开发中的标准代码。在实际开发中，具体项目有具体的代码设计和书写规范，请读者重点理解思路。

一般来说，游戏中会有一个单例脚本来管理角色的一些数据信息，这个脚本不继承自Mono类，属于一个单例脚本，主要起一个存取并管理数据的作用。我们先创建一个脚本，命名为Player.cs，然后打开它，删掉它对Mono类的继承，删掉自带的Start和Update函数（因为这两个函数没有用处了）。

首先，需要在这个脚本中声明一个静态的实例，并书写一个公共的获取静态实例的静态方法，让它像单例一样运作，方便我们存取数据。

然后在这个脚本中声明5个私有变量：playerName（string类型，表示角色名称）、maxHp（float类型，表示最大生命值）、currentHp（float类型，表示当前生命值）、maxMp（float类型，表示最大魔法值）、currentMp（float类型，表示当前魔法值），我们将名称变量暂时赋值为“张三”，另外4个私有变量全部都临时赋值为1000，并为它们书写公有的设置/获取函数。

注1：声明私有变量，然后书写共有的设置/获取函数，不但可以保护访问的安全性，还可以在以后可能出现的复杂的数据存取操作中更加方便。

注2：因为UISlider组件的进度值是一个浮点数，我们也需要将进度显示到小数点后两位的百分比，所以，这里对生命HP和魔法MP全部都采用float类型，以方便使用。

具体代码如下（Player.cs）：

```
using UnityEngine;
using System.Collections;
public class Player {
    //声明静态实例
    private static Player Instance;
    //获取静态实例
    public static Player GetInstance()
    {
        if (Instance == null)
        {
            Player ins = new Player();
            Instance = ins;
            return Instance;
        }
        else
        {
            return Instance;
        }
    }
    //声明私有变量
    private string playerName = "张三";
    private float maxHp = 1000;
    private float currentHp = 1000;
    private float maxMp = 1000;
    private float currentMp = 1000;
    //公有的设置获取方法
    public void Set_playerName(string value)
```

```
{
    playerName = value;
}
public string Get_playerName()
{
    return playerName;
}
public void Set_maxHp(float value)
{
    maxHp = value;
}
public float Get_maxHp()
{
    return maxHp;
}
public void Set_currentHp(float value)
{
    currentHp = Mathf.Min(value, maxHp);
}
public float Get_currentHp()
{
    return currentHp;
}
public void Set_maxMp(float value)
{
    maxMp = Mathf.Min(value, maxMp);
}
```

```

public float Get_maxMp()
{
    return maxMp;
}
public void Set_currentMp(float value)
{
    currentMp = value;
}
public float Get_currentMp()
{
    return currentMp;
}
}

```

现在我们需要做的是制作UI脚本，创建一个新的脚本，命名为 **RoleStateUI**，然后打开，在其中声明UI组件的引用并书写逻辑，为了方便，在Update中书写一个“扣血”和“加血”的功能，来验证我们的逻辑。

我们先来看看代码：

```

using UnityEngine;
using System.Collections;
public class RoleStateUI : MonoBehaviour {
    public UILabel playerNameLabel;
    public UISlider hpSlider;
    public UISlider mpSlider;
    public UILabel hpPercentLabel;
    public UILabel mpPercentLabel;
    void Start () {

```



```
//检查引用的健全性，这是一个查错的好习惯
if (playerNameLabel == null)
{
    Debug.Log("角色名称Label的引用丢失了！");
}
else
{
    //为角色名称赋值
    playerNameLabel.text =
Player.GetInstance().Get_playerName();
}
if (hpSlider == null)
{
    Debug.Log("角色血条的引用丢失了！");
}
if (mpSlider == null)
{
    Debug.Log("角色魔法条的引用丢失了！");
}
if (hpPercentLabel == null)
{
    Debug.Log("角色血量文字的引用丢失了！");
}
if (mpPercentLabel == null)
{
    Debug.Log("角色魔法量文字的引用丢失了！");
}
```

```

    }

    void Update () {
        //书写一个测试用的方法，来扣血和加血
        //按下A，则扣血182，按下B加血114
        if (Input.GetKeyDown(KeyCode.A))
        {
            Player.GetInstance().Set_currentHp(Player.GetInstance().Get
            _currentHp() - 182);
        }
        if (Input.GetKeyDown(KeyCode.B))
        {
            Player.GetInstance().Set_currentHp(Player.GetInstance().Get
            _currentHp() + 114);
        }
    }

    void LateUpdate()
    {
        //在每一帧更新完成后之后，更新血条和魔法量
        float hppercent = Player.GetInstance().Get_currentHp() /
        Player.GetInstance(). Get_maxHp();
        hpSlider.value = hppercent;
        hpPercentLabel.text = (hppercent * 100).ToString(".00") + "%";
        float mppercent = Player.GetInstance().Get_currentMp() /
        Player.GetInstance(). Get_maxMp();
        mpSlider.value = mppercent;
        mpPercentLabel.text = (mppercent * 100).ToString(".00") + "%";
    }

```

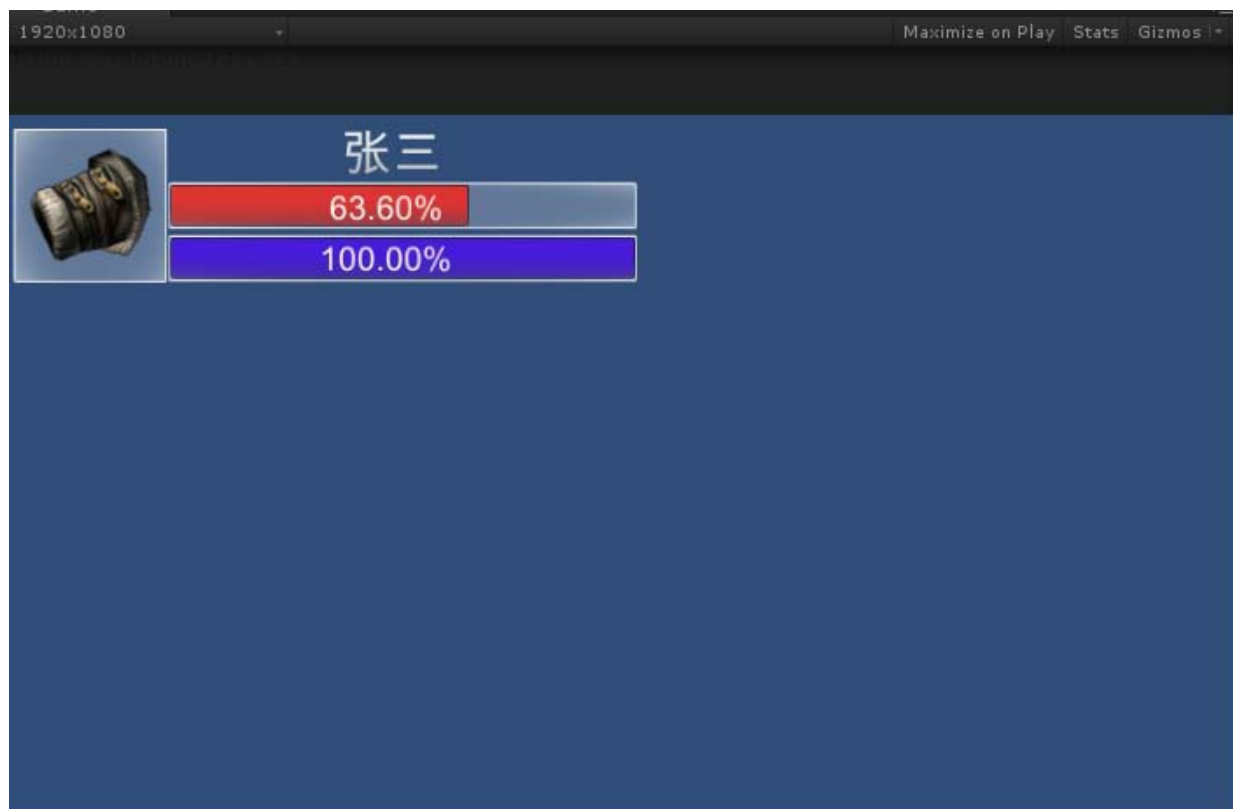
```
}
```

我们在脚本中首先声明了UI组件的引用，然后在**Start**中给角色名称赋值，并且检查了所有的组件引用是否存在，这是一个写代码的好习惯，因为，一旦因为不存在引用而导致了**Bug**我们就能第一时间知道问题所在。然后我们在**Update**中写了两个测试用的功能：按**A**扣血和按**B**加血。最后，我们在**LateUpdate**中更新角色的血量和魔法量的UI显示。

我们将代码挂在**MainUI**（也就是**UIRoot**）上，然后将引用的各个组件依次拖上去赋值给它。运行游戏，在运行状态下，我们按**A**会扣血，按**B**会加血，血量进度条和文字会跟着一起实时显示，如图8.3所示。

根据这个模块，我们还可以扩充更多的功能：比如头像图标也会根据职业或者角色来显示不同的，再比如血量低于一定百分比之后角色名称和血条会变色等。

角色状态栏一般都和其他的一些主界面菜单一起构成了“主界面”，这个界面一般是在游戏中长期存在的，如果是永久存在，可以在主界面的脚本的开始部分写上这样一句代码 **DontDestroyOnLoad(this.gameObject)**，就可以让它在场景中永远存在。如果一部分地方存在一部分地方不存在，可以考虑将主界面做成一个预设体，然后在进入新场景时，判断这个场景是否需要主界面，以此决定是否需要加载**Prefab**。



▲ 图8.3

## 8.2 场景加载的进度条界面制作

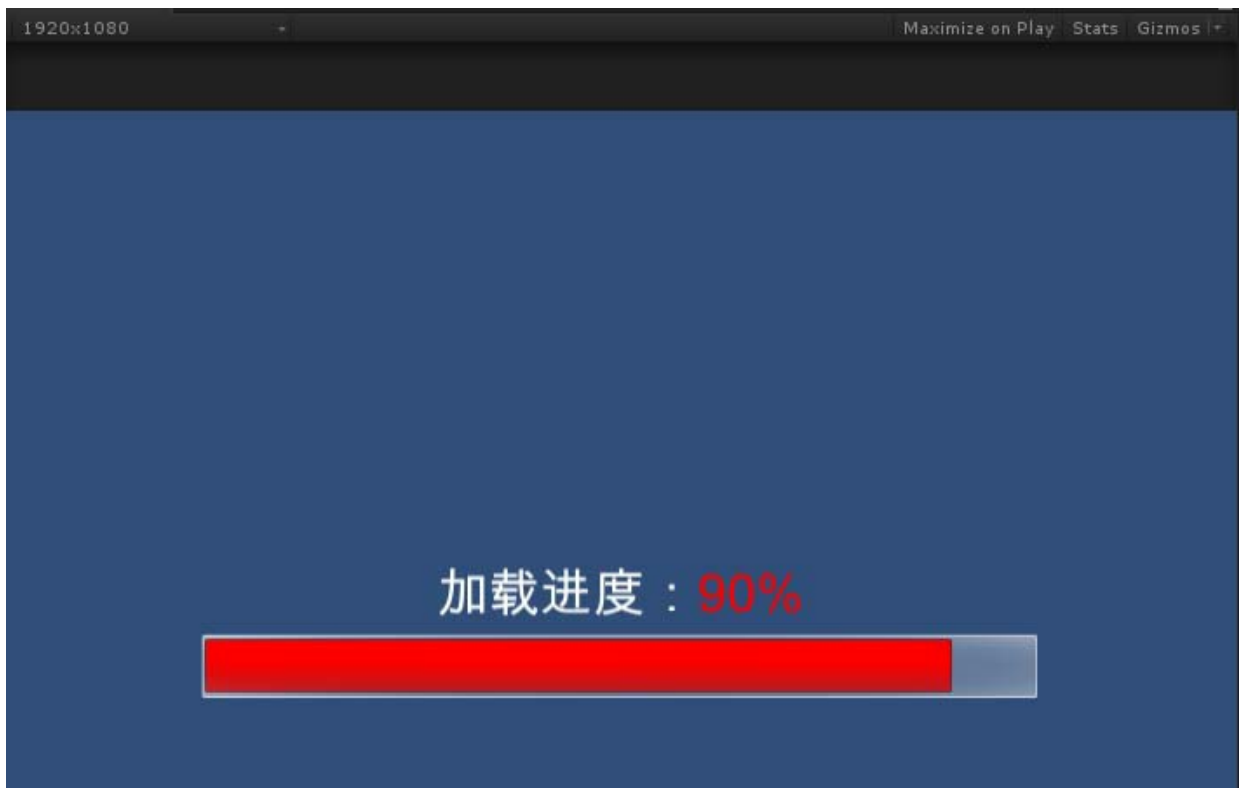
### 8.2.1 为什么要做这个界面

在游戏中，我们经常会涉及场景切换，Unity切换场景时，会清空旧场景的内存，然后将新场景的资源全部加载到内容中。而新场景的内容如果稍大，则会在切换场景的时候出现卡顿，这个卡顿是由于在加载资源到内存造成的。而在稍微大型点的项目中，经常会有资源量很大的场景，这个时候我们需要做一个加载界面，来告诉玩家我们正在加载新场景，并告知玩家加载的进度如何了。

这就是我们所谓的Loading界面，如图8.4所示。

## 8.2.2 异步加载的概念

在NGUI中，加载新场景分为同步加载和异步加载，所谓的同步加载就是加载完了新场景才会继续执行别的事。所谓的异步加载，就是程序会在后台加载新场景，在此过程中还可以执行其他的事情。



▲ 图8.4

同步加载: `Application.LoadLevel();`

异步加载: `Application.LoadLevelAsync();`

为了显示这个进度条，我们可以在加载场景的时候，先加载进一个专门的Loading场景，因为这个场景只有Loading信息，所以加载非常快，然后在这个场景中立马进行新场景的加载，并实时地显示加载进度。

需要注意的是，异步加载虽然是后台在加载，但是，因为程序在对内存进行着大量的操作，所以依然会导致有一定的卡顿。

### 8.2.3 制作一个单独的加载界面场景

我们新创建一个场景，将场景命名为LoadingScene。然后在这个场景中制作图8.4所示的加载界面。

制作步骤如下。

#### 1. 创建UI

在这里我们可以创建一个2DUI在场景中，并将UIRoot命名为LoadingUI。

#### 2. 设定分辨率

这里我们假定项目标准分辨率是1920×1080，于是将Game视窗调为1920×1080，然后将UIRoot设为缩放模式，Height为1080。

#### 3. 创建锚点

因为进度条是靠底部的，在这个案例中假设进度条需要靠近底部定位。所以，先在UIRoot下创建一个Anchor，创建方法为Unity顶部NGUI菜单，依次选择Creat→Ancho。然后我们将Anchor的锚点设置为Bottom，并将Anchor重命名为Anchor\_Bottom。

#### 4. 创建进度条

我们先在Anchor\_Bottom下创建一个Sprite，命名为LoadingSlider，将它的尺寸调整到我们所需要的底板的大小，然后调整好在屏幕中的位置。为它附加一个UISlider的组件，这个Sprite将作为进度条的底板使用。

在LoadingSlider下创建一个子物体Sprite，命名为Foreground，这个Sprite将会作为显示进度的上层条使用，调整它的大小和位置。然后，将这个Sprite物体拖动到LoadingSlider物体的UISlider组件的Foreground设置项中。

在LoadingSlider下创建一个子物体Label，命名为ProgressLabel，它将作为进度的文字显示，将Label的文本先设定为：“加载进度：

100%”，然后调整文本的字体大小和颜色。

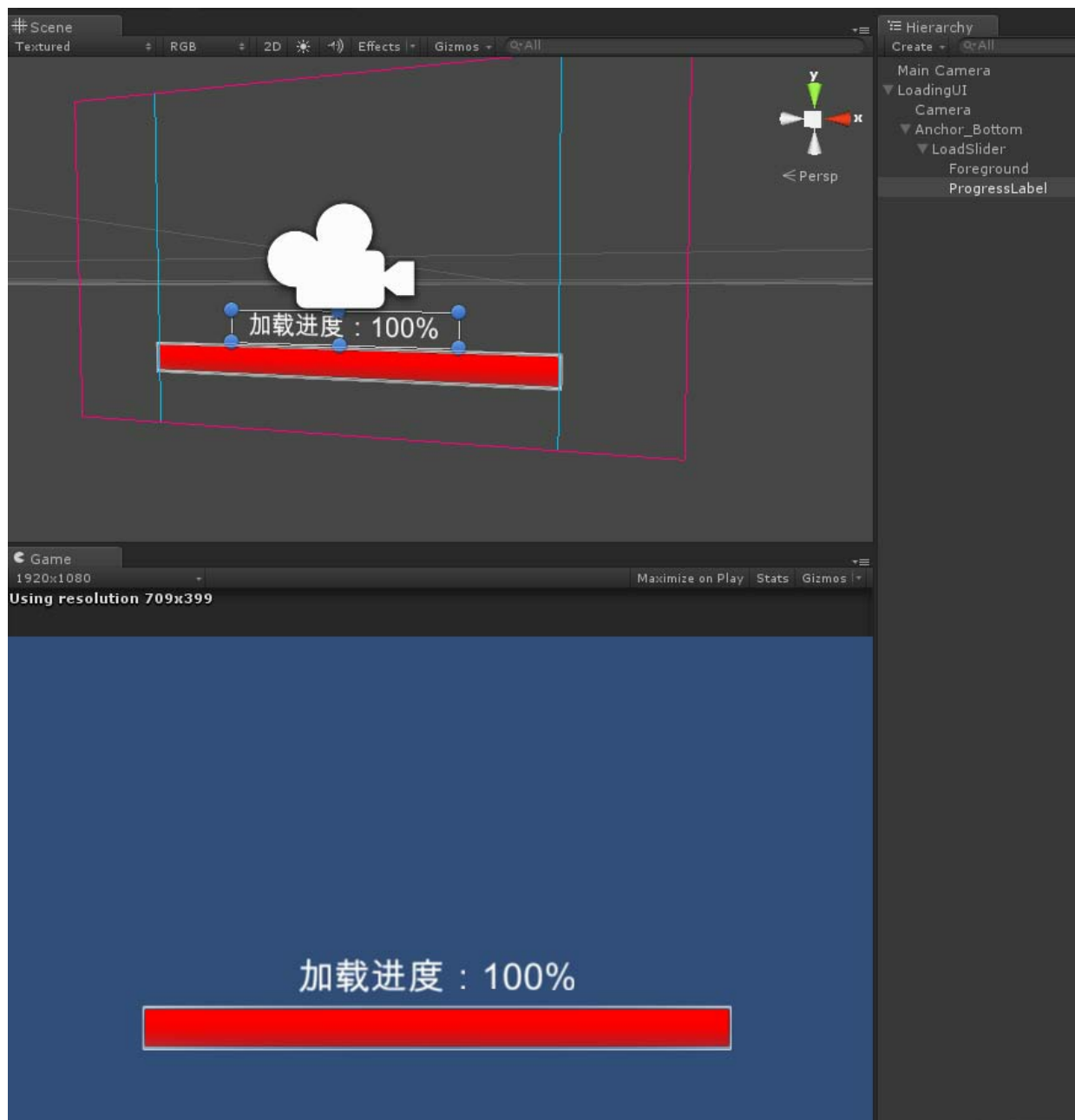
制作好后，如图8.5所示。

这样我们的加载场景和界面基本就算制作完成了。

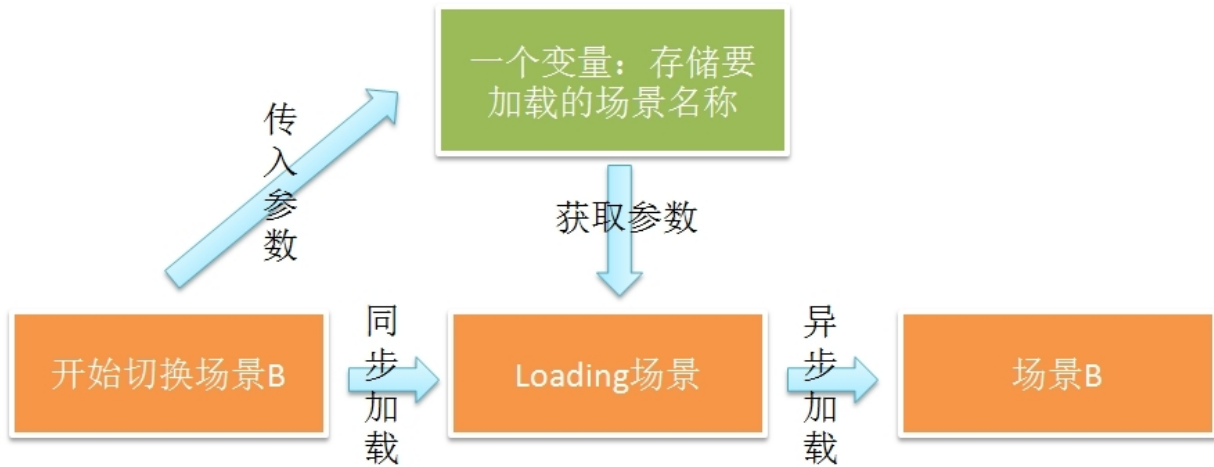
#### **8.2.4 设计并编写代码**

我们在加载新场景时，利用Loading场景非常小的特点，先同步加载Loading场景，在加载的同时，将真正要加载的场景名称作为一个参数传入到一个全局变量（可以用单例或者静态变量来实现）中保存起来，然后Loading场景加载后，立即获取这个变量，立即开始加载真正要加载的场景，并实时显示加载进度。这里的代码逻辑如图8.6所示。





▲ 图8.5



▲ 图8.6

我们准备用一个静态变量来临时保存要加载场景的名称。我们新创建一个 C#脚本，命名为LoadingScene.cs，并双击打开，写上如下代码：

```
using UnityEngine;
using System.Collections;

public class LoadingScene : MonoBehaviour {
    //先声明一个静态字符串变量来保存要加载场景的名称
    public static string LoadingName;
    //引用UI组件
    public UISlider slider;
    public UILabel label;
    //声明一个异步进度变量
    AsyncOperation asyn;
    void Start () {
        //检查组件引用的正确性
        if (slider == null)
        {
            Debug.Log("进度条组件丢失！");
        }
    }
}
```

```

    }
    if (label == null)
    {
        Debug.Log("进度显示文字丢失！");
    }
    //进入这个场景就立即协程加载新场景
    StartCoroutine("BeginLoading");
}

void Update () {
    //更新UI
    slider.value = asyn.progress;
    label.text = "加载进度: [FF0000]" + (slider.value *
100).ToString(".00") + "%";
}
//加载目标场景的函数
IEnumerator BeginLoading()
{
    asyn = Application.LoadLevelAsync>LoadingName);
    yield return asyn;
}
//设计一个封装好的静态函数
public static void LoadNewScene(string value)
{
    LoadingName = value;
    Application.LoadLevel("LoadingScene");
}
}

```

在代码中可以看到，声明了一个静态变量LoadingName来保存要加载的目标场景的名称，然后封装了一个静态函数 LoadNewScene()来供切换场景时调用。如果需要切换场景 B，我们就调用：

```
LoadingScene. LoadNewScene("B");
```

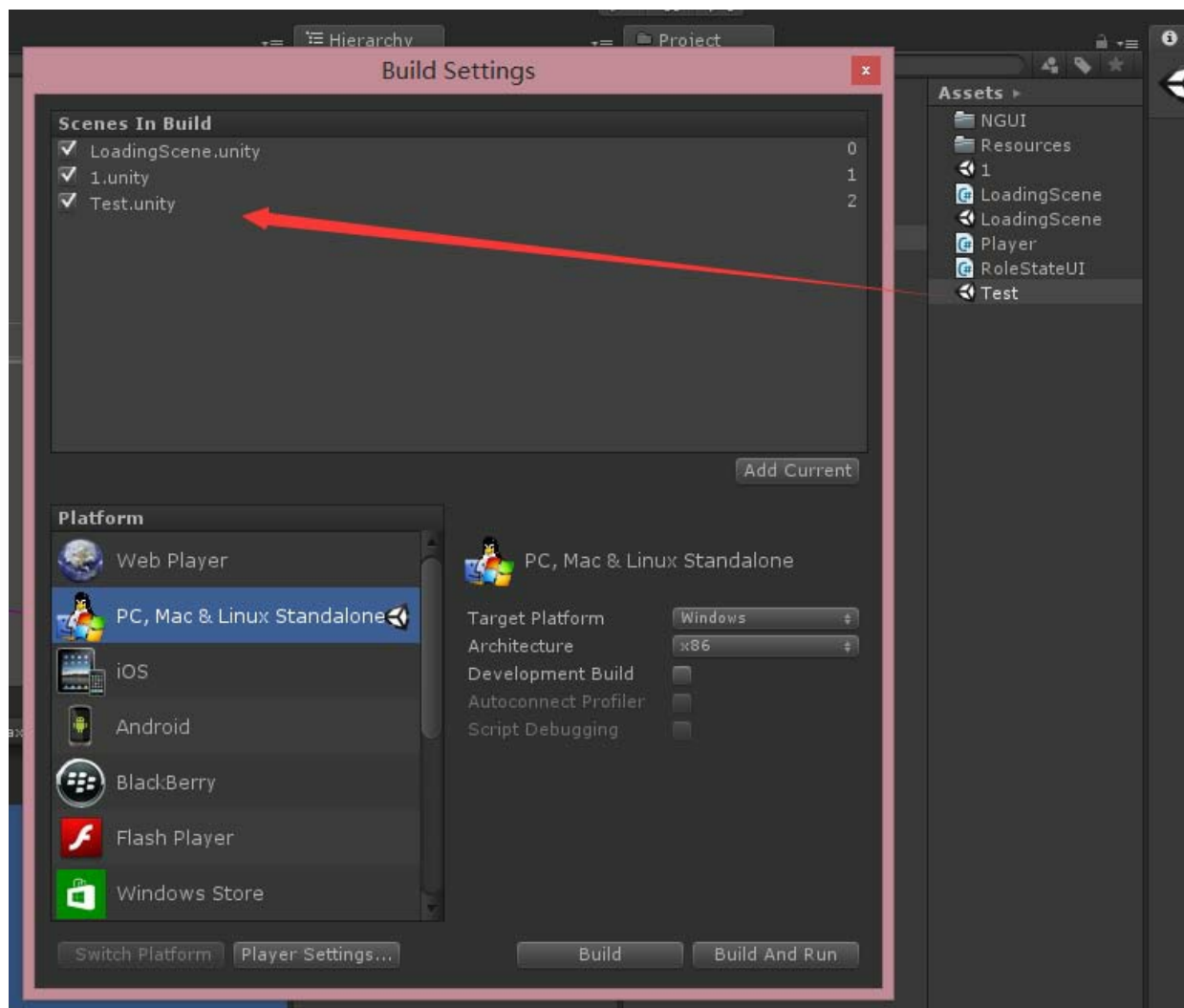
然后的逻辑是立即将 B 保存到静态变量 LoadingName 中，并且加载 Loading 场景，当Loading场景被加载进来之后，在Start函数中就开启了一个协同程序去执行BeginLoading方法并加载目标场景，同时返回 asyn异步实例，然后再利用这个asyn在Update中实时地更新UI信息。

因为需要加载进度后面的百分比数字显示为红色，所以，给赋值时加入了Label的文本关键字：[FF0000]，当Label识别到这个关键字，它就不会显示这个关键字，而是将关键字后面的文本内容全部变成红色。

当异步的进度达到100%之后，就相当于后台进行加载完成了，程序会自动切换场景到目标场景中。

一次完整的加载流程就到这里结束。因为这个过场加载需要目标场景的内容足够的大，这样加载起来Loading进度看着才明显，所以，大家可以自己进行尝试，本文就不配图标示了。制作Loading的原理虽然如此，但是代码并不一定非要如此组织，大家可以在理解了这个原理之后，根据自己的喜好自由地去进行代码组织。

特别注明一点：Loading 场景和需要加载的目标场景，一定要已经放到了 BuildSetting 中才可以进行加载！在Unity顶部菜单，依次选择 File → BuildSetting，弹出界面，然后将场景文件拖动到界面中即可，如图8.7所示。

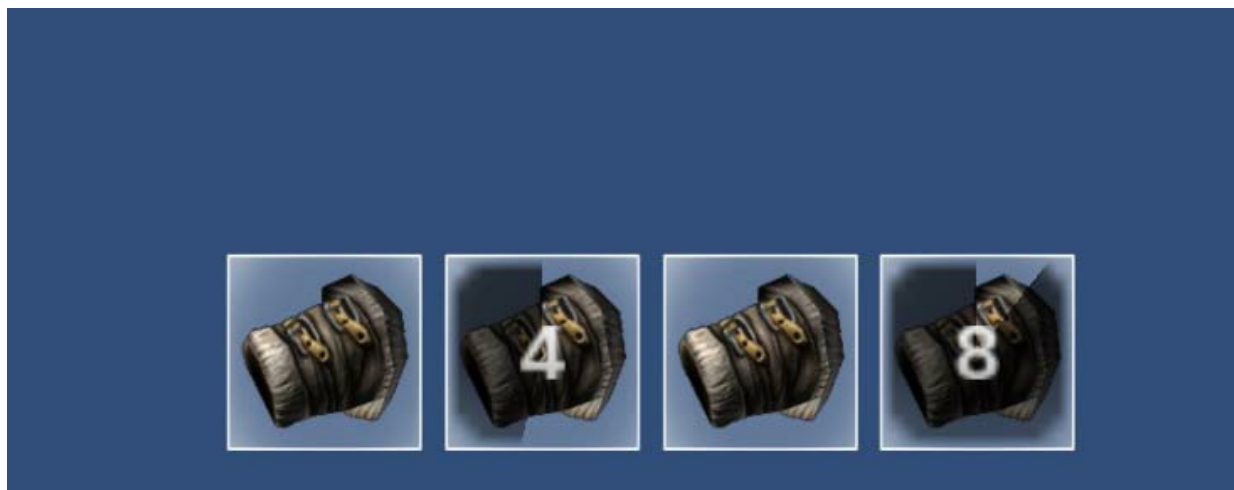


▲图8.7

## 8.3 技能快捷栏的制作

### 8.3.1 示意图和需求分析

技能快捷栏，是大部分游戏中都存在的一种界面，不管是RPG游戏、策略游戏、竞技游戏等，都有技能快捷栏的影子。如图8.8所示为一个标准的简易技能快捷栏。这也是我们这节要完成的案例目标。



▲ 图8.8

从图8.8中，我们先来罗列以下制作这个案例的需求。

- (1) 一共4个技能格子，需要显示图标。
- (2) 单击格子后，技能开始CD，并出现CD进度的遮罩。
- (3) CD时需要实时显示CD剩余秒数。
- (4) CD过程中无法再次单击技能。
- (5) 假定项目标准分辨率为1920×1080。
- (6) UI需要永远处于屏幕底部位置。

### 8.3.2 设计并制作UI

我们新创建一个场景，命名为SkillUIScene，然后将在这个新场景中完成技能快捷栏的制作，读者也可以在其他场景中进行制作，原理是一样的。先来了解一下制作步骤。

#### 1. 创建UI

然后在场景中创建一个2DUI，将UIRoot命名为MainUI。

#### 2. 设置分辨率

每当新创建UIRoot后，一定不要忘记设置项目开发标准分辨率。在这里我们将UIRoot设为缩放模式，Height为1080。并将Game视窗的

分辨率调整为1920×1080。

### 3. 设置锚点

每当在新的UIRoot下制作UI时，除了分辨率以外，一定不要漏掉锚点的考虑。因为这直接关系到屏幕适配的问题。

在本文的案例目标中，我们看到技能快捷栏偏于屏幕下方，在UIRoot下创建一个Anchor，设置锚点位置为Bottom，并将Anchor物体重命名为Anchor\_Bottom。

### 4. 制作技能格子UI

因为技能快捷栏有多个格子，所以，先在Anchor\_Bottom下创建一个Panel来管理所有的技能格子，将这个Panel重命名为SkillPanel。

然后在Panel下创建一个Sprite，命名为SkillTemplate，这个Sprite将作为技能格子的底子。因为这个格子要承担被单击的任务，所以，为它Attach一个BoxCollider。然后在这个Sprite下创建一个Sprite子物体，命名为Icon。

在这个案例中，我们是假设4个技能都已经装备上了，所以，提前把Icon设置好，可以从Atlas中选择一个图标作为技能图标。

然后继续在SkillTemplate这个物体下创建一个Sprite子物体，命名为Cover，选择一个白色的色块图片作为Cover，首先我们将Cover的颜色调为纯黑色，并让它半透明。然后，将它的模式设为Filled，并将Filled Dir设为Radial360。这样在技能CD时，这个遮罩就可以转圈表示CD进度了。

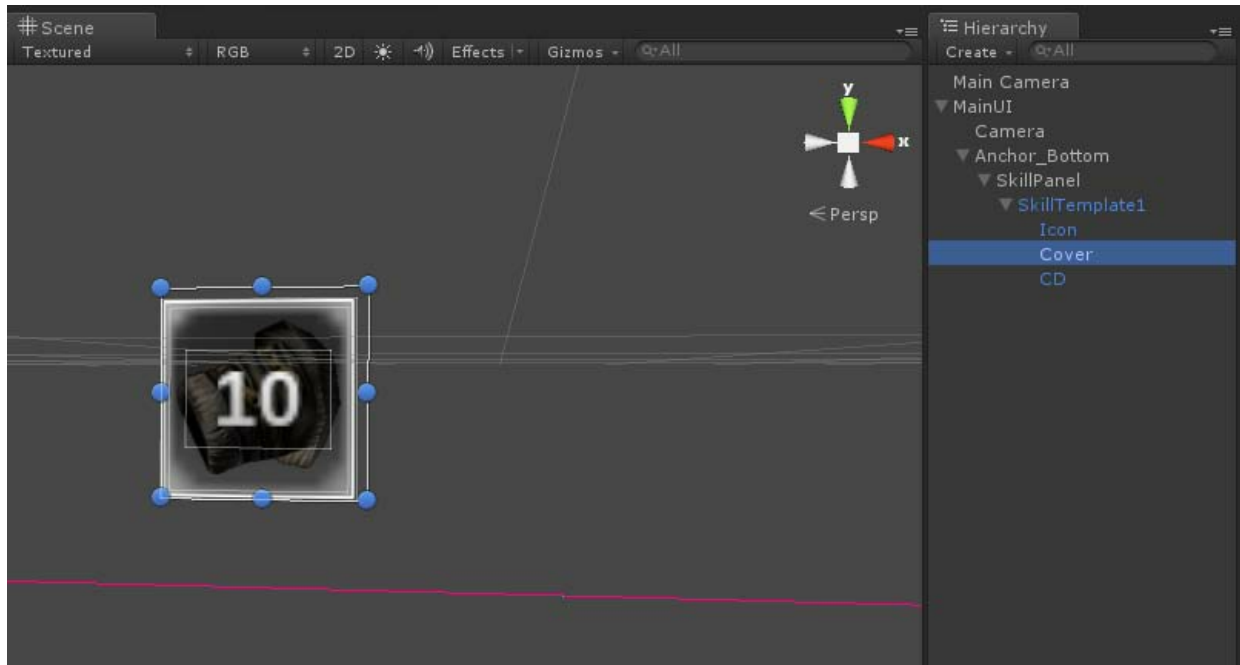
最后我们在SkillTemplate下再创建一个Label子物体，命名为CD，用来表示技能剩余的CD时间的文本。

做到这一步，调整好各个控件的尺寸大小到合适的大小，并调整SkillTemplate（这个物体目前就已经代表了一个完整的技能格子）到合适的位置。然后要进行一件很重要的事情：Depth深度检查，因为技能界面层级较多，而且层级关系不能混乱，所以，一定不要忘记检查

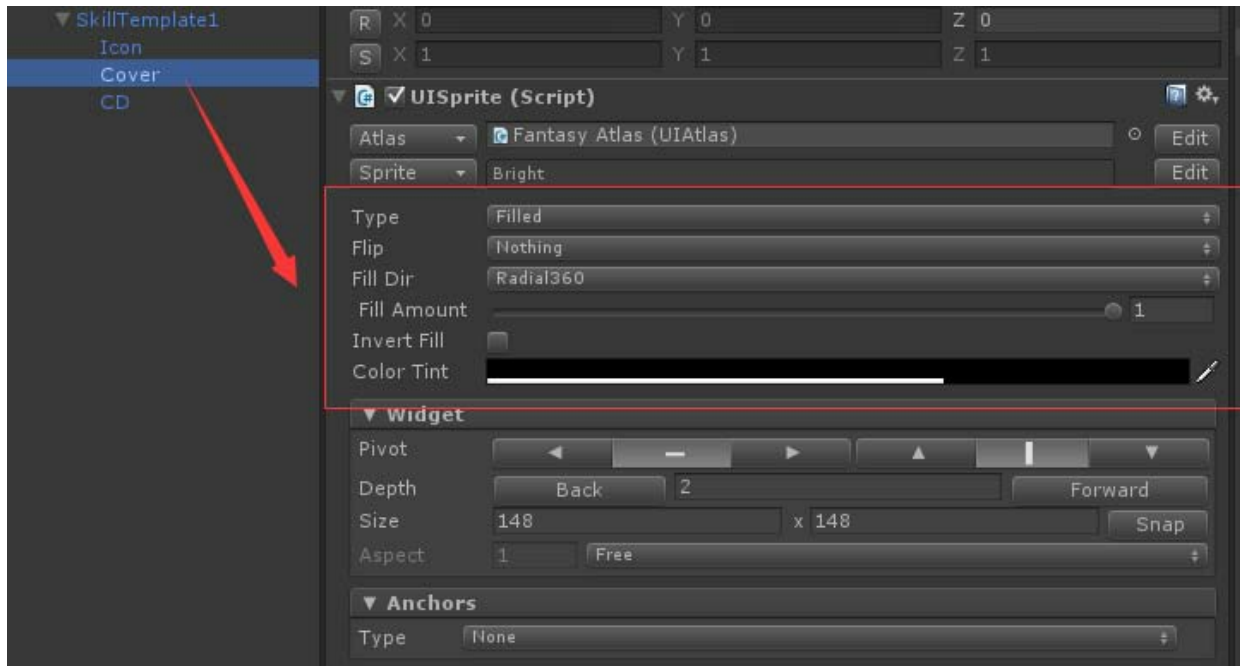


Depth，从技能格子的需求来看，深度关系应该为：CD的Label >技能遮罩的Sprite >技能图标Sprite >技能底板的Sprite。

图8.9表示了制作好的UI界面雏形。图8.10表示了Cover遮罩的Sprite组件设置。



▲ 图8.9

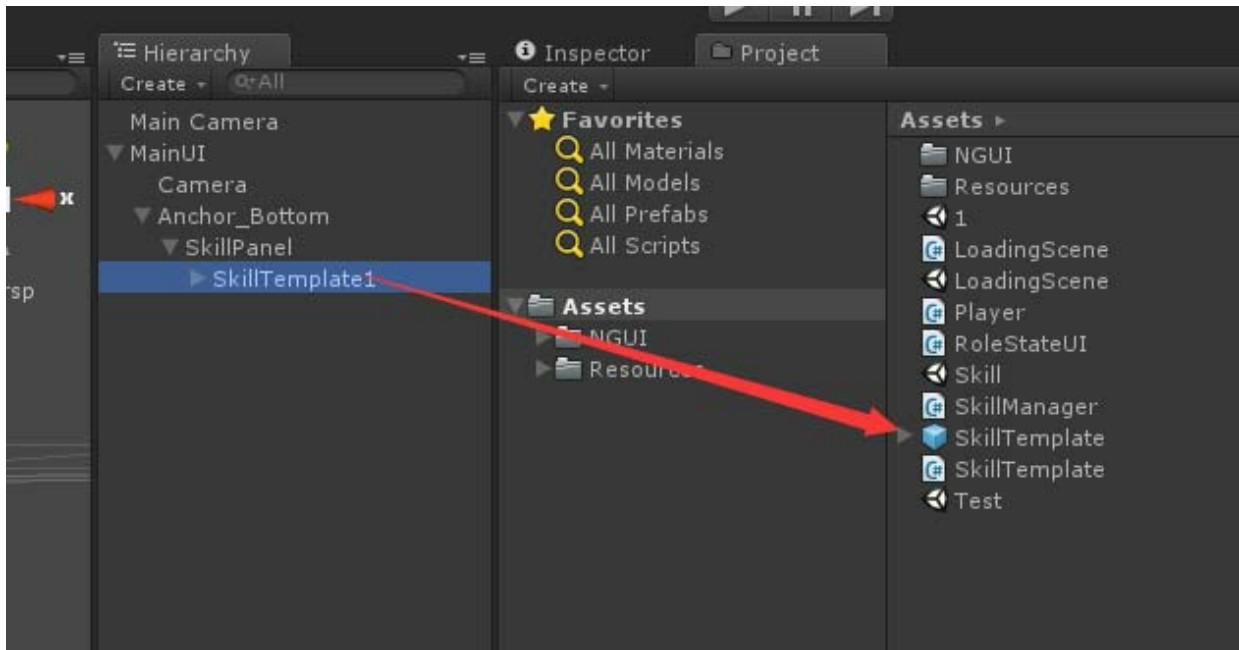


▲图8.10

### 5. 利用预设，完成技能快捷栏

从案例需求分析中可以得知：每一个技能格子其实都是一模一样的。所以，这里我们就可以使用 **Prefab** 预设体来制作技能格子，也就是将刚才制作好的一个技能格子保存为一个预设体，然后利用复制更多预设体来完成技能快捷栏的制作。这样做除了可以更快捷、更方便外，还有一个更重要的意义：修改的时候，只需要修改一个技能格子，然后在**Inspector**面板中**Apply**即可让修改内容应用到所有的技能格子上。如果不使用预设体，那么修改的时候，就得每个格子都修改一遍，如果游戏中有30个技能格子，修改任何东西都需要修改30遍，这是一个极不科学的设计。

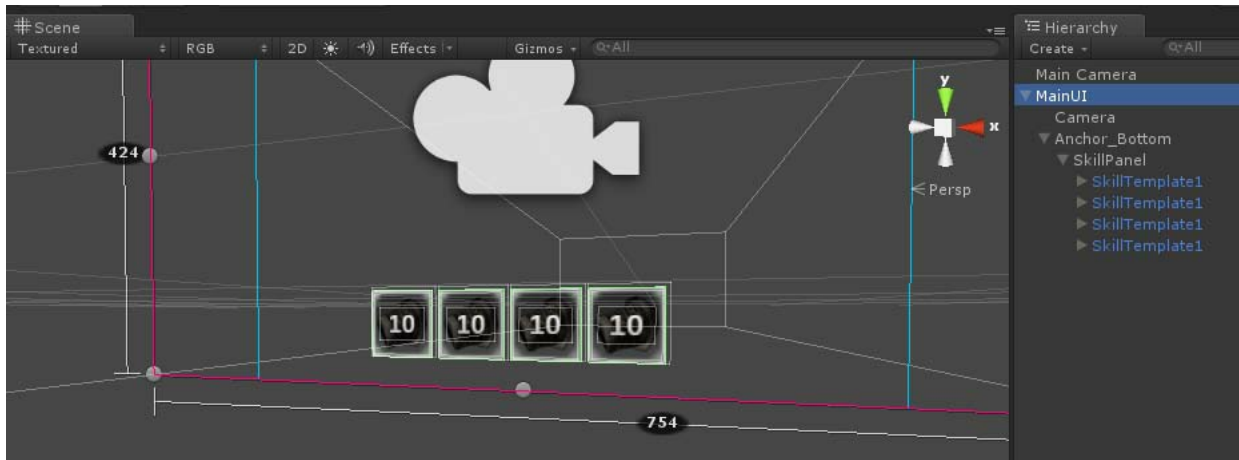
我们现在将制作好的技能格子直接拖动到**Project**面板中，它就会自动保存为一个预设体，如图8.11所示。



▲图8.11

然后我们在**SkillPanel**下，再拖进去3个预设体，一共构成4个预设体，调整好这4个预设体的位置，就完成了技能快捷栏的制作，如图

8.12所示。



▲ 图8.12

### 8.3.3 设计并编写代码

在开始本章节的代码讲解之前，先声明一些注意事项。

(1) 在一般游戏中，技能快捷栏里面的技能数据（也就是这个技能格子代表的是哪个技能）是需要传入进去的，在这里为了方便讲解，让读者明白原理，就直接将技能数据录入到格子上。

(2) 本章节的代码只负责讲明技能快捷栏UI的原理，所以很多东西是固定的，并不是游戏开发中的标准代码，建议大家重点学习技能UI的工作原理，在实际开发中以项目组的需求为准。

(3) 为了方便讲解，在本章节做了一个前提假设：技能快捷栏上已经有4个技能了。

首先，需要创建一个 C#脚本，命名为 **SkillTemplate**。这个脚本的主要用途是管理单个技能格子的UI逻辑，我们将会把这个脚本挂到技能格子的预设上，让每一个技能格子都拥有这么个脚本。

打开SkillTemplate脚本，写上以下代码：

```
using UnityEngine;  
using System.Collections;
```

```

public class SkillTemplate : MonoBehaviour {
    //声明技能的CD时长
    public float CDTime = 10.0f;
    //声明UI控件的引用
    UISprite icon;
    UISprite cover;
    UILabel cdLabel;
    //记录剩余CD时间的变量
    float leftCD = 0;
    //标示技能是否CD好的变量
    bool isReady = true;
    void Start () {
        //找到UI控件的引用
        icon = transform.FindChild("Icon").GetComponent<UISprite>();
        cover = transform.FindChild("Cover").GetComponent<UISprite>
    );
        cdLabel = transform.FindChild("CD").GetComponent<UILabel>
    );
        //引用的检查
        if (icon == null)
        {
            Debug.Log("技能图标引用丢失！");
        }
        if (cover == null)
        {
            Debug.Log("技能CD遮罩引用丢失！");
        }
    }
}

```

```
    if (cdLabel == null)
    {
        Debug.Log("CD文本引用丢失！");
    }
    //初始化UI
    InitSkillUI();
}
//初始化这个技能格子
void InitSkillUI()
{
    if (isReady)
    {
        cover.fillAmount = 0;
        cdLabel.text = "";
    }
}
//检查技能CD的函数
void CheckCD()
{
    if (isReady == false)
    {
        if (leftCD > 0)
        {
            leftCD -= Time.deltaTime;
            cover.fillAmount = leftCD / CDTime;
            cdLabel.text = ((int)leftCD).ToString();
        }
    }
}
```

```

        else
        {
            isReady = true;
            cover.fillAmount = 0;
            cdLabel.text = "";
        }
    }
}
//单击技能格子的触发事件
void OnClick()
{
    if (isReady)
    {
        isReady = false;
        leftCD = CDTime;
    }
    else
    {
        Debug.Log("技能还没有CD好");
    }
}

void Update () {
    //在Update中检查CD
    CheckCD();
}
}

```

在代码中我们可以得到以下思路。

(1) 首先需要在这个技能格子类中，知道这个格子代表的技能是什么，这样才能知道它的一些与UI有关的关键属性，比如CD时间。在本章中，为了演示方便，直接声明了一个公共变量**CDTime**来表示技能的CD时间。

(2) 因为技能格子的 UI 全部都是这个格子的子物体，所以，没有必要去声明 **Public** 的UI组件引用然后去拖动，可以很方便地直接在代码中使用**transform.FindChild**的方法来找到这些UI组件的引用。

(3) 在脚本运行时，要先初始化这个技能，让它显示出正常的状态。

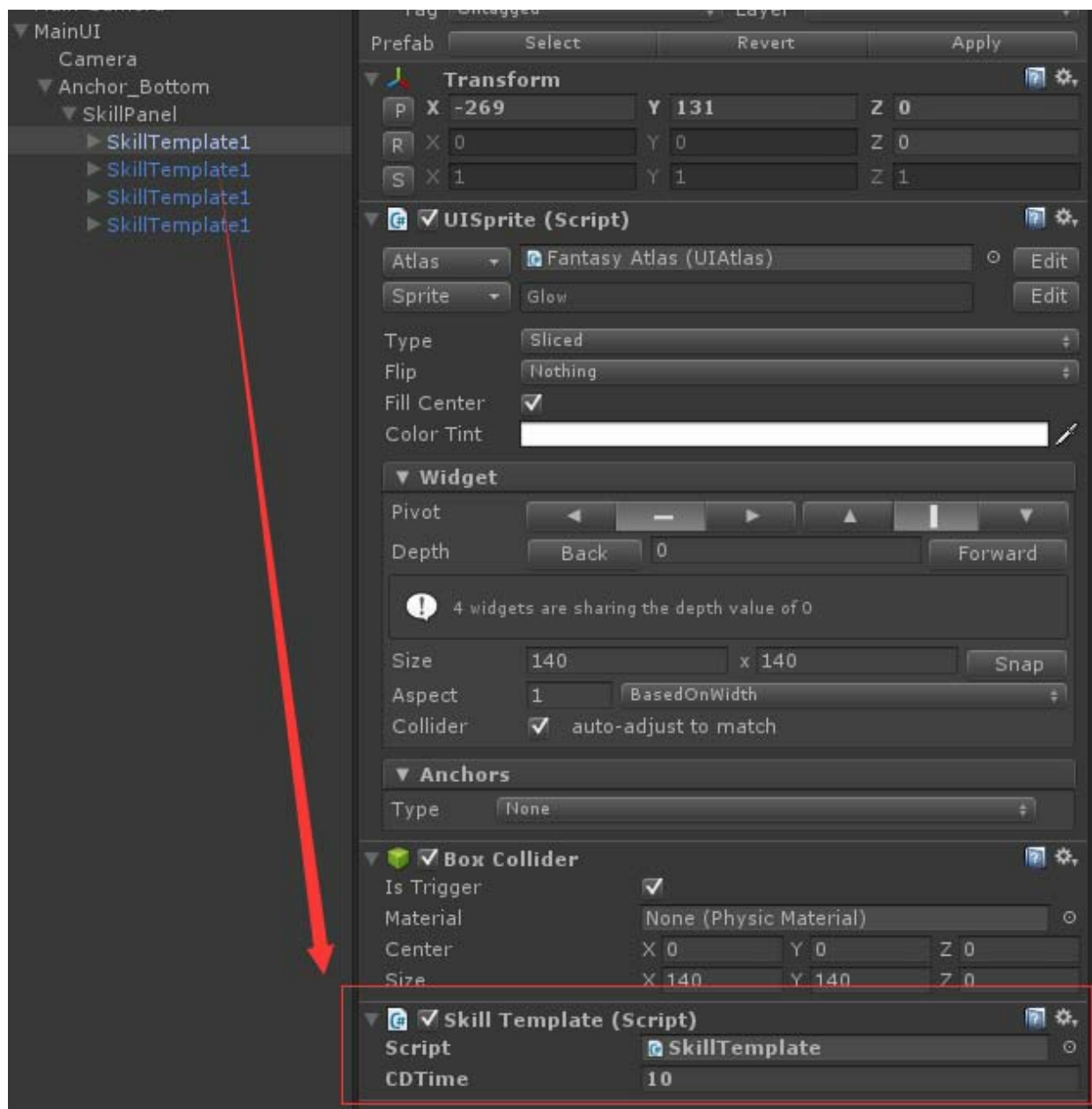
(4) 我们需要写一个使用技能的方法，单击技能格子就会调用这个方法，可以为技能格子添加一个**UIButton**组件，并在**Button**里面设置单击的回调事件，但是，这样做很麻烦。所以，直接在技能脚本中使用了事件监听方法：**void OnClick()**。只要技能格子被单击了，就会调用这个格子身上的脚本中的**OnClick**方法。

(5) 单击技能格子后，技能的**isReady**变为了**false**，于是技能开始进入了**CD**状态。

(6) **Update**函数中加入了**CD**的检测，如果技能进入了**CD**，立即开始执行走**CD**时间的逻辑，并实时、正确地显示在UI上。

我们将脚本挂到技能格子 **SkillTemplate** 的预设体上，单击 **Apply**按钮，让每一个技能格子都拥有了这个脚本，可以在这个脚本的组件中去修改它的**CD**时间，如图8.13所示。





▲ 图8.13

然后运行游戏，就可以看到4个技能格子已经处于准备好的状态，单击格子之后，技能遮罩开始转圈，CD文本实时地显示着技能还有多少秒就CD完成，并且在CD过程中再次单击技能，会在控制台打印出一句话：技能还没有CD好。当技能CD完成之后，技能又回到了初始状态。

## 8.4 角色头顶血条的跟随

### 8.4.1 角色头顶血条的跟随分析

在很多RPG游戏或者竞技游戏中，在玩游戏中操控着一个角色进行游戏，这个角色头顶一般都跟着一个血条，不管角色跑到什么地方，这个血条在视觉上，看上去就好像永远都处于角色的头顶上方一样。这就是常见的角色头顶血条跟随。

我们知道，要让一个物体相对于另一个物体永远都保持相对位置不变，最简单的做法就是：让这两个物体形成父子物体关系，这样子物体将永远和父物体保持相对的位置不会改变。但是，如果头顶的血条当作角色的子物体来做有几个问题。

(1) 角色是会转换方向的，这会导致血条也跟着旋转。

(2) 在3D游戏中，角色是有透视关系的，而血条作为UI一般都是没有透视关系的。也就是说，人物跑远了会变小，但是血条作为承载着重要信息的UI并不会随着人物跑远而变小。

当然，如果项目的需求很简单，可以使用父子物体的方法解决固然更好。如果不能简单地通过父子物体来解决，可以用以下思路来制作：让血条在UI摄像机中实时地和角色的屏幕坐标进行同步。

### 8.4.2 制作血条的UI

血条的UI很简单，就是一个UISlider，我们在第一个案例中就已经专门讲过Slider的做法，这里就不多描述制作过程了。

需要注意的是，很可能在很多角色上都使用这个血条，比如角色头顶有血条，各种各样的怪物NPC头顶也可能会有血条，所以，我们需要将这个血条保存为一个Prefab预设体。

我们创建一个新场景，创建一个2DUI，将UIRoot命名为MainUI之后，在MainUI下面制作好一个血条，将它保存为一个预设，命名为BloodBar。然后我们即将来学习一下怎样让它跟随角色的头顶。

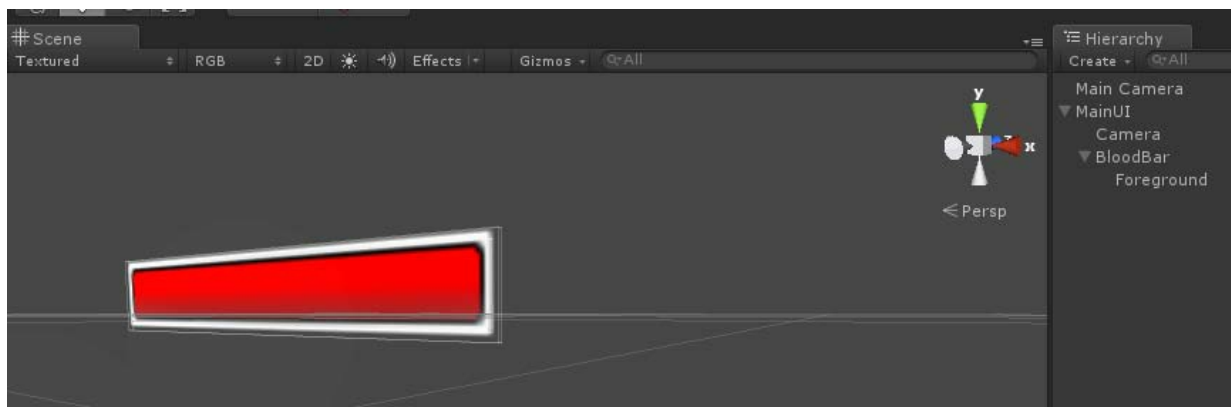
注意：

- (1) 血条如果需要动态加载或者多人共用，一定要做成预设。
- (2) 血条不能放在任何一个Anchor下面，因为血条不能根据屏幕进行适配，它只能和目标角色保持相对位置不变。

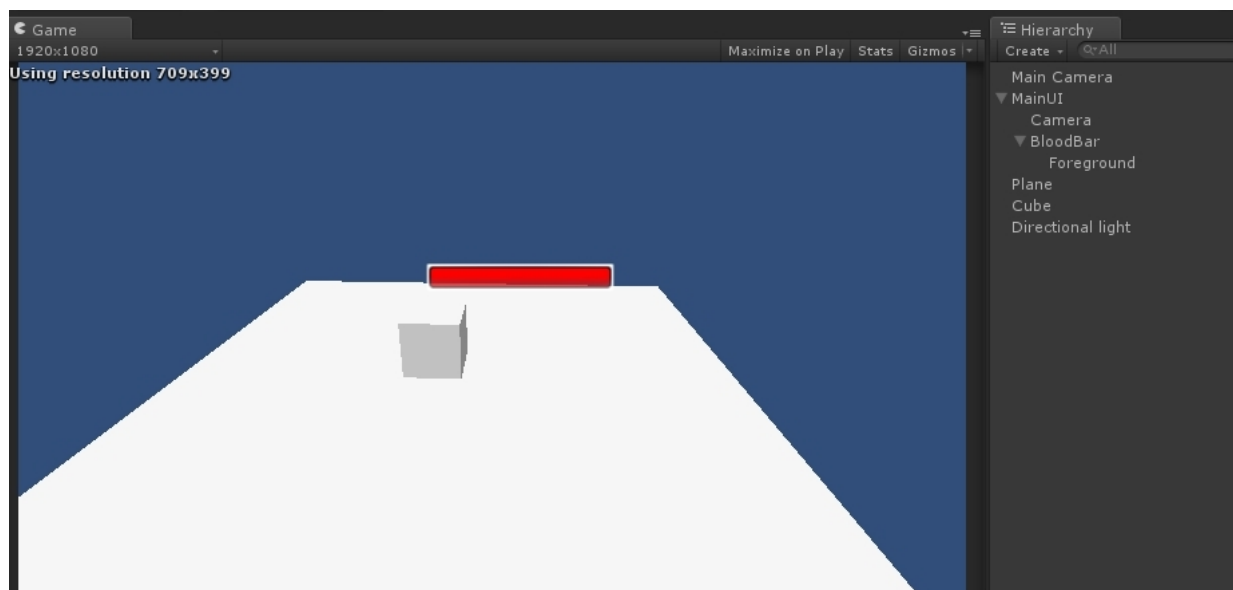
图8.14表示了一个做好的血条。

### 8.4.3 设计并编写代码

首先，我们在场景中创建一个Plane平面和一个Cube立方体（创建方式为Unity顶部菜单，依次选择GameObject → Create Other → Cube/Plane），这个平面表示地面，这个立方体表示角色，然后调整好相机角度，创建一个平行光，如图8.15所示。



▲ 图8.14



▲ 图8.15

我们创建一个 C#脚本，命名为 **RoleControl**。将会用这个脚本来控制角色移动，并让 **UI**跟随角色并进行绑定。

具体代码如下：

```
using UnityEngine;
using System.Collections;

public class RoleControl : MonoBehaviour {
    //声明血条的引用
    public GameObject bloodUI;
    //声明角色移动的速度
    float speed = 5.0f;
    void Start () {
        //检查引用的正确性
        if (bloodUI == null)
        {
            Debug.Log("血条引用丢失了！");
        }
    }
}
```

```

}

void Update () {
    //用WASD来控制角色
    if (Input.GetKey(KeyCode.W))
    {
        transform.position += Vector3.forward * Time.deltaTime *
speed;
    }
    if (Input.GetKey(KeyCode.A))
    {
        transform.position += Vector3.left * Time.deltaTime * speed;
    }
    if (Input.GetKey(KeyCode.S))
    {
        transform.position += Vector3.back * Time.deltaTime * speed;
    }
    if (Input.GetKey(KeyCode.D))
    {
        transform.position += Vector3.right * Time.deltaTime * speed;
    }
}

void LateUpdate()
{
    //每一帧渲染完成后，实时更新血条的位置
    //得到主摄像机中人物头顶1米处在屏幕上的坐标
    Vector3 pt = Camera.main.WorldToScreenPoint(new
Vector3(this.transform.position.x, this.transform.position.y + 1,

```

```
this.transform.position.z));
```

//然后将它转换为NGUI摄像机的坐标

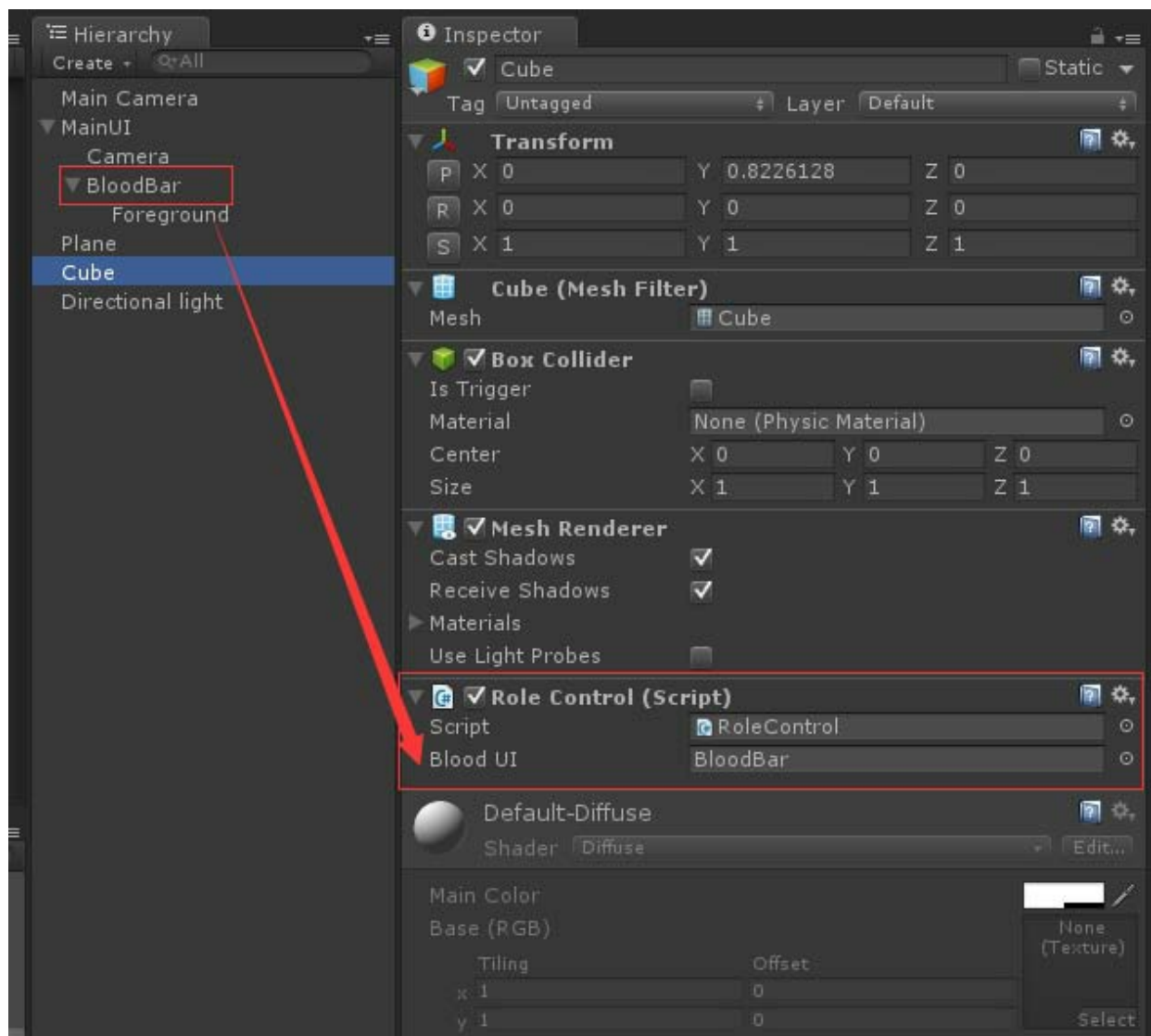
```
bloodUI.transform.position
```

```
=UICamera.FindCameraForLayer(bloodUI.layer).camera.ScreenToWorldPoint(new Vector3(pt.x,pt.y, 1));
```

```
}
```

```
}
```

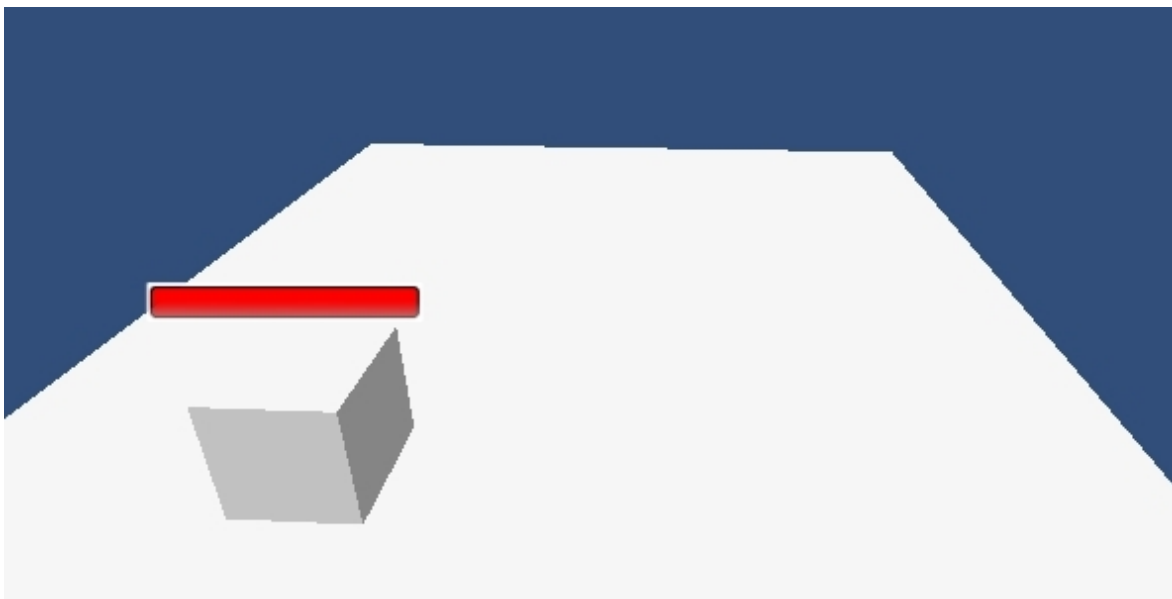
将脚本挂到Cube上去，然后将血条UI拖到RoleControl组件的引用上，如图8.16所示。



▲ 图8.16

然后运行游戏，按下键盘上的 **WASD** 可以控制这个立方体前后左右移动，可以看到血条也会跟随这个立方体移动，并永远处于立方体头顶跟随，并且UI的大小永远不会改变，如图8.17所示。

核心原理就是先将角色头顶1米处的坐标转换为屏幕坐标，将这个屏幕坐标转换为UI相机的世界坐标，然后将这个坐标实时赋予UI，视觉上就达到了跟随的效果。



▲ 图8.17

## 8.5 NGUI多语言切换的实现

### 8.5.1 什么是本地化

所谓的本地化，可以理解为游戏的多语言切换管理。因为游戏可能销往世界各地，而世界各地的语言又不尽相同，我们希望玩家能够根据自己的母语来选择语言，例如，美国玩家选择英文进行游戏、中国大陆玩家选择简体中文进行游戏，这就是本地化。



本地化并不是指制作多个游戏包发往不同的地方，而是所有人用的都是同一个游戏，然后在里面可以自由地选择语言种类。

NGUI自带的本地化系统目前支持所有UILabel的文本进行语言切换。

### 8.5.2 NGUI本地化的原理

旧版本的NGUI可能需要Localization对象来设置不同的语言文件。而在新版本的NGUI中，进一步简化了本地化的操作。现在以NGUI 3.7.2 版本为例进行讲解。

NGUI 的本地化系统，需要用到 3 个东西：语言选择组件、文本读取组件、语言配置文件。

#### 1. LanguageSelection组件

这个组件的路径为：

AddComponent → NGUI → Interaction → LanguageSelection。

这个组件附加到一个UI控件上之后，会自动在该控件上生成一个UIPopupList组件用来选择语言。当运行游戏时，这个PopupList会被LanguageSelection组件应用，自动读取语言配置文件，从而变成一个语言选择的PopupList。

图8.18演示了NGUI的LanguageSelection语言选择控件的组件结构，它必须包含：一个Sprite、一个 BoxCollider、一个 LanguageSelection、一个 PopupList（会被 LanguageSelection自动生成，也可以手动添加）。

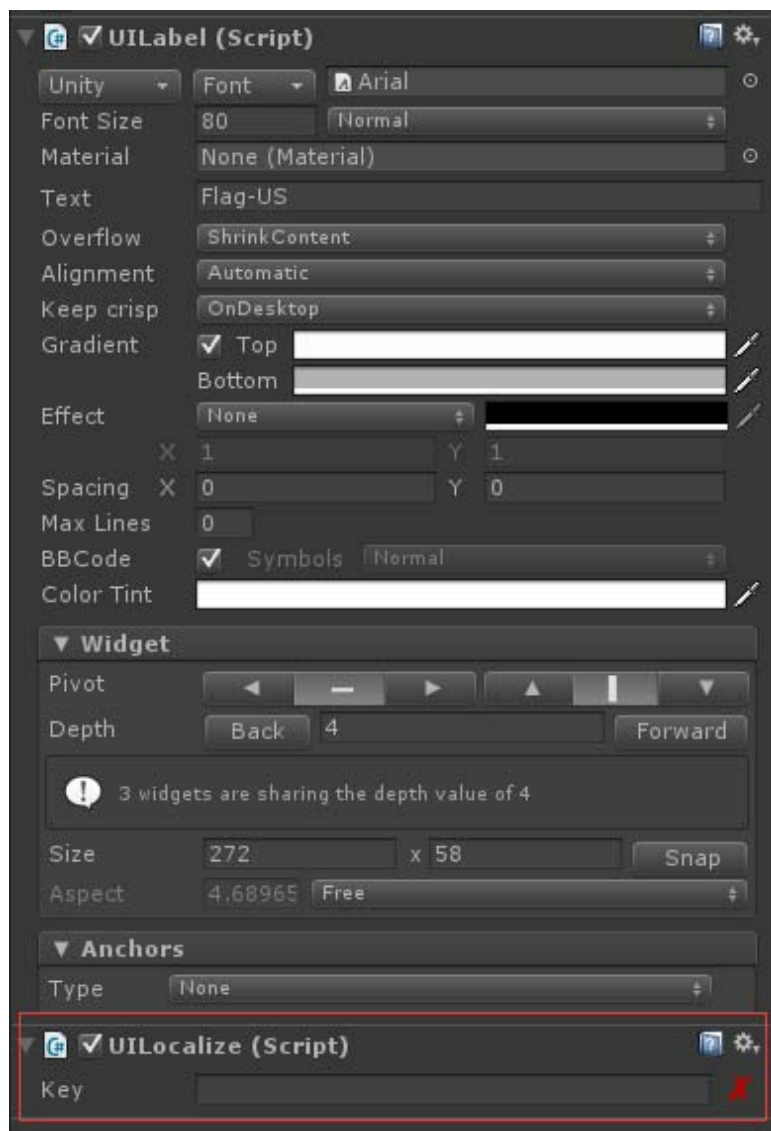


▲ 图8.18

值得注意的是，如果UIPopupList和LanguageSelection组件一起存在于控件上，那么这个 **PopupList** 控件将会强制变成一个语言选择的下拉菜单。也就是说，此种情况下， **UIPopupList** 中录入的各种选项都将无效，在游戏运行时，里面的选项会自动变成语言配置文件中的语言选项。

## 2. UILocalize组件

任何一个需要语言切换的UILabel，都必须附加上这个组件，如图8.19所示。



▲ 图8.19

这个组件需要填入一个“Key”，这个 Key就是这个 Label 在不同语言中的索引。我们会有一个语言配置文件，它的结构如表8.1所示。

表8.1

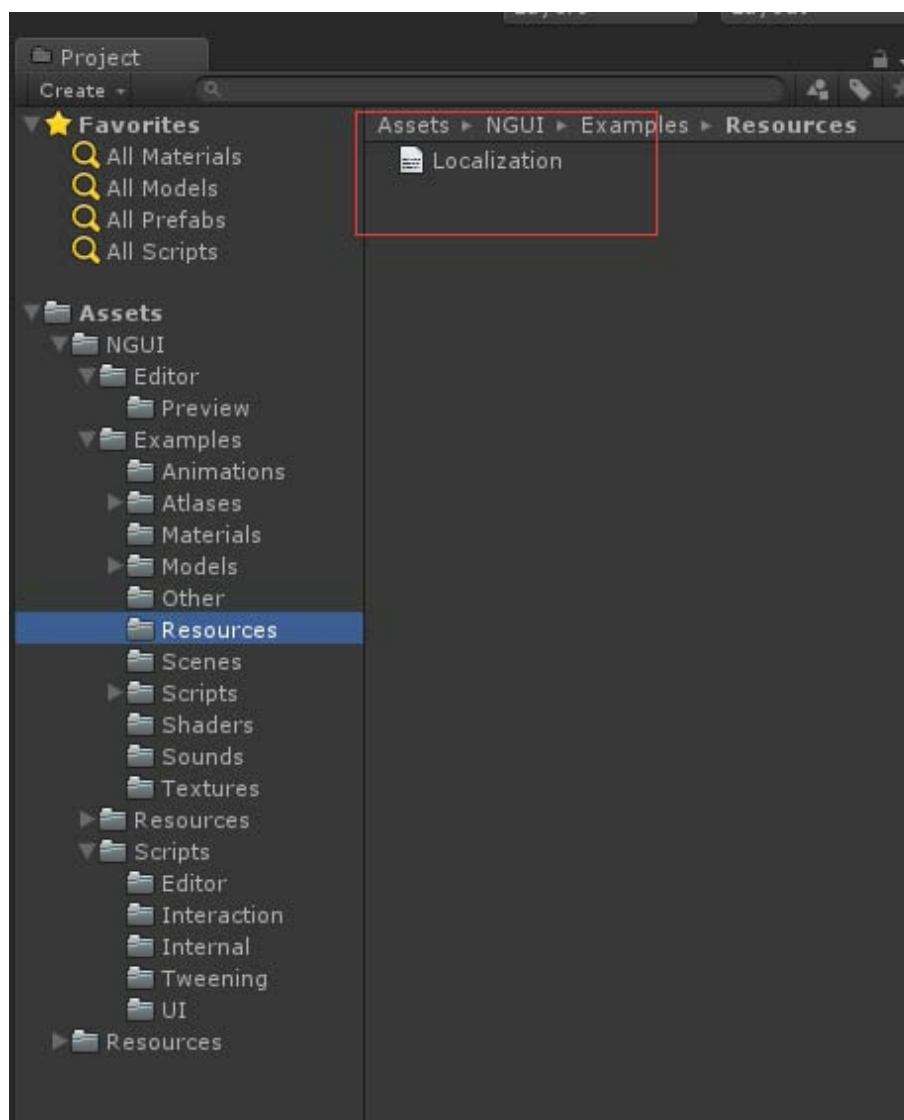
Key	中文	英文
NameLabel	名称	Name
Label2	这是一个新文本	This is a new label
Label3	专为中文使用	For English

然后在UILabel控件上挂上UILocalize之后，填入Key，例如填入Key为Label2，那么当切换到中文时，这个UILabel会根据Key从刚才的表格中寻找显示的值，会显示“这是一个新文本”；切换到英文之后，UILabel会显示“This is a new label”。

### 3. Localization.txt文本文件

这个文本文件就是刚才提到的语言配置文件，NGUI中有一个供调用的文本文件，我们只需要修改它即可，路径为：

Assets/NGUI/Examples/Resources/Localization。注意，这是一个txt文件，路径如图8.20所示。



### ▲ 图8.20

我们打开这个文件之后，看到如图8.21所示的内容。

```
KEY,English,Français
Flag,Flag-US,Flag-FR
Language,English,Français
Info,Localization example,Par exemple la localisation
Sound,Sound,Son
Music,Music,Musique
Desc,English localization,La localisation française
```

### ▲ 图8.21

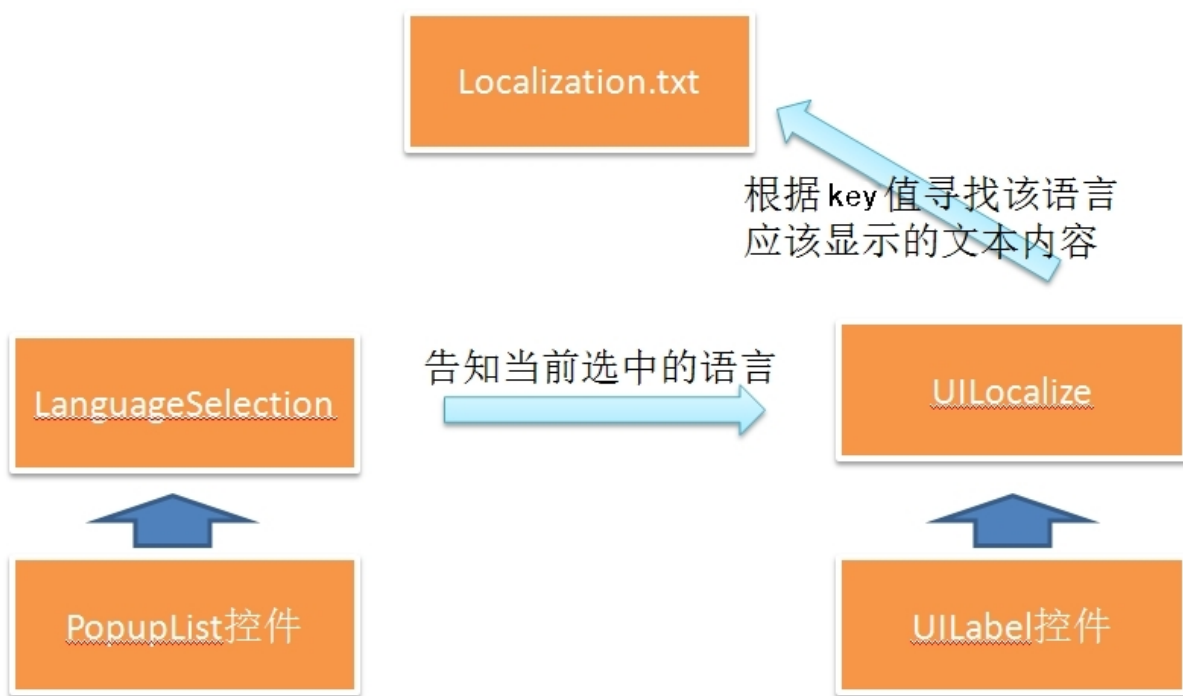
这个文本文档其实和刚才讲的表8.1是一模一样的结构，只不过它是用逗号（一定要是英文输入法下的逗号）隔开的，可以将这个文件中NGUI目前自己写的范例部分翻译成一个表格，可以更容易理解，如表8.2所示。

表8.2

Key	English	Français
Flag	Flag-US	Flag-FR
Language	English	Français
Info	Localization example	Par exemple la localisation

这样就可以很清晰看到这个文本文档其实就是一个表格格式。凡是带有UILacolize组件的UILabel都将会从语言配置文件中，根据Key值寻找当前选择的语言模式下应该显示的文本内容。

这3个元素之间的关系如图8.22所示。



▲ 图8.22

### 8.5.3 本地化案例演示

我们新创建一个场景，来进行一个本地化案例的演示。

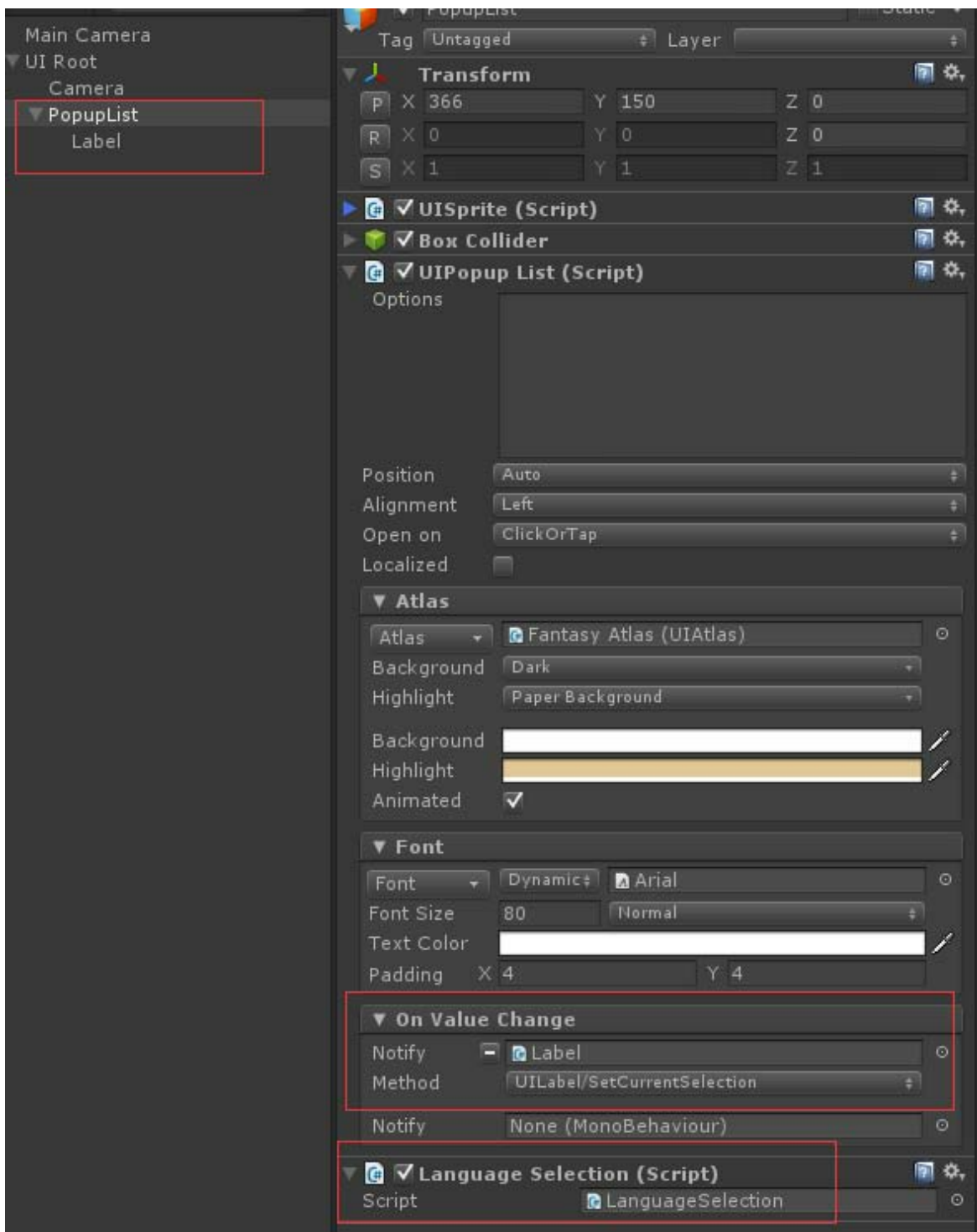
首先，归纳好需要多语言切换的Label有哪些，然后完善Localization.txt配置文件，我们从Assets/NGUI/Examples/Resources/Localization中打开这个文件，将里面的内容改为图8.23所示的内容。

```
KEY, English, 中文
Name, Lucy, 露西
Sex, Female, 女性
Age, Ten, 十岁
```

▲ 图8.23

然后在场景中创建一个Sprite控件，为它Attach一个BoxCollider组件。并且为它添加一个LanguageSelection组件（会自动生成UIPopupList组件），然后在自动生成的PopupList组件中设置好Background和Highlight的图片。我们在这个PopupList物体下创建一个Label子物体，并拖动到PopupList的OnValueChanged事件回调中，让它实时显示这个PopupList当前选中的选项。PopupList的选项内容不需要录入，因为它在游戏运行时会自动被LanguageSelection组件设置好，如图8.24所示。





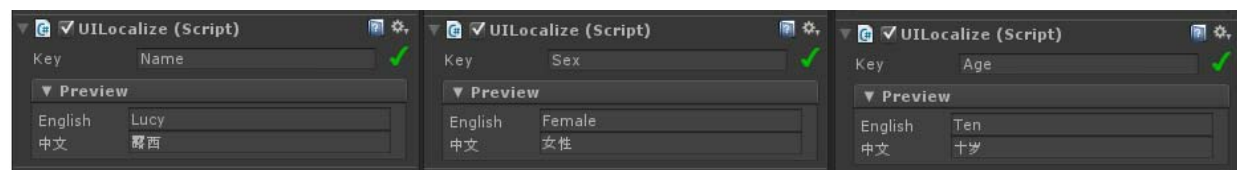
▲ 图8.24

其次我们在场景中制作3个Label，如图8.25所示。



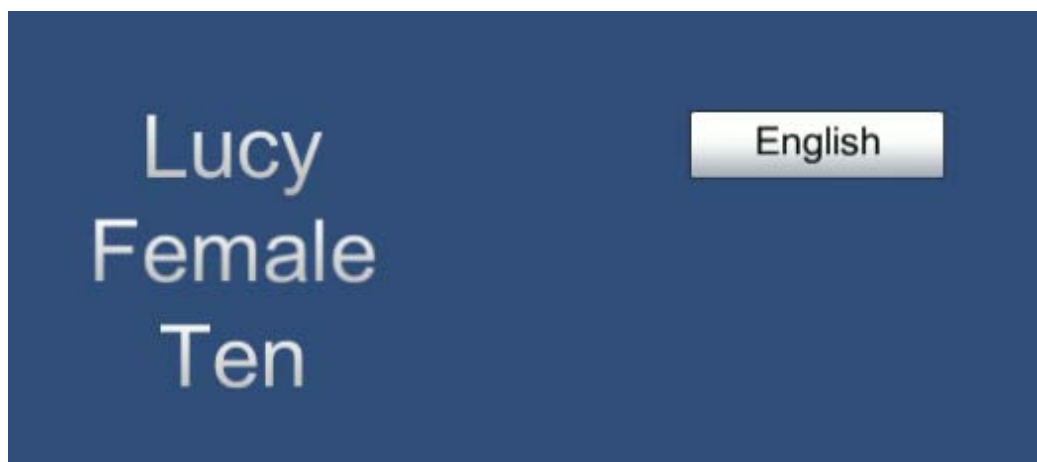
▲ 图8.25

我们为这3个Label分别添加一个UILocalize组件，它们的Key值分别设置为图8.23中我们填写的Key值：Name、Sex、Age，如图8.26所示。



▲ 图8.26

我们运行游戏，可以看到图8.27所示的画面。



▲ 图8.27

单击选择语言的PopupList会出现图8.28所示的画面。



▲ 图8.28

我们从语言选择中选中中文，会出现图8.29所示的画面。



▲ 图8.29

# 第9章 常见疑难问题解答

## 9.1 关于NGUI版本问题

NGUI作为世界上最流行的Unity界面制作插件，一直以来都更新迭代非常频繁，也许很多人对NGUI的版本很困扰，下面我们就来统一了解一下与版本有关的一些问题。

（1）我们可以在Project窗口中选中NGUI文件夹，可以看到有一个ReadMe的版本说明文件，这个文件标明了该NGUI是哪个版本，打开之后里面写了NGUI最近几次版本更新的内容。

（2）NGUI 的版本，只有过一次革命性的更新，就是从NGUI 2.X版本更新到NGUI 3.X版本，NGUI 3.0以后的版本只要版本号相差不远，功能几乎都一样。比如说NGUI 3.6.0 及以后的版本，区别都很小。需要特别说明的是，NGUI的版本不管怎么更新，它的知识点都大同小异，不用为版本担心。

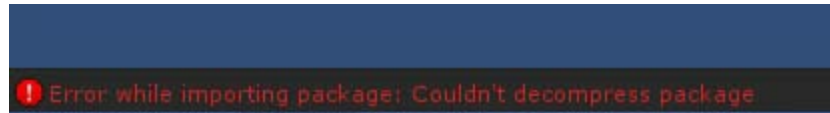
（3）NGUI版本更新之后，如果要使用新版本的NGUI，可以将旧版本的NGUI删掉，导入新版本NGUI，然后检查一下已经做好的UI有没有组件丢失的情况，如果有重新赋值一下组件。

## 9.2 导入NGUI资源包出错

(1) 如果已经导入了NGUI的一个版本的资源包了，此时再次导入NGUI的资源包会有什么结果？

答：会根据路径替换掉同名文件，并额外导入新的文件。

(2) 导入解压后，并没有弹出资源包内容预览窗口，而是在Unity编辑器窗口底部报出了一行如图9.1所示的错误，怎么办？



▲ 图9.1

答：这是因为将NGUI资源包放在了一个带有中文名称的文件夹路径下，Unity导入任何带有中文路径的资源包时，都会弹出这个错误导致无法导入。请将要导入的资源包放在一个没有任何中文的路径下再进行导入。

## 9.3 如何创建两个UIRoot

NGUI中严格意义上是不能直接创建两套UI的，如果已经创建了一个UI，不管是3D还是2D的UI，那么，Unity顶部NGUI菜单中创建UI的选项都会变灰无法再创建。

但是，也许会碰到需要两套UI的情况，这时可以把场景中已有的UI的UIRoot组件关闭（也就是把组件激活的勾去掉），然后再进行创建即可。

## 9.4 如何让粒子在界面上正确显示

NGUI中，渲染的层级关系是由Depth决定的，但是，最本质的还是由渲染的Render Queue决定的，这是一个Shader中常见的参数。在NGUI中，每一个Panel上也有一个RenderQ的设置项，RenderQ越高的将会越上层显示。

粒子系统的RenderQ一般是3000，所以，如果我们希望粒子处于两个Panel之间，只需要将其中一个Panel的RenderQ改为StartAt模式，将值设为3000以下的值，然后将另一个Panel的RenderQ设为3000以上的值，就可以让粒子在两个Panel之间显示了。

当然，如果只需要让粒子显示在最上层，最简单的办法就是加入一个摄像机，给这个粒子设置一个单独的 Layer，让新加入的摄像机只渲染粒子所在的 Layer，将这个摄像机的 Clear Flags 设为Depth Only，然后将渲染的Depth 值设为最高的即可。

## 9.5 为什么在父物体上增加透明度动画，子物体没有跟着变化

也许有时候我们会这样做：用一个父物体来管理 N 多个子物体控件，比如让 5 个 Sprite都处于同一个父物体下，这样就可以通过对父物体进行位移实现所有控件一起位移等效果。

在这种情况下，我们让父物体位移动画、放缩动画等都能带着子物体一起运动。但是，透明度动画却有时候不会带着子物体一起变得透明，这是为什么呢？

因为透明度，可以理解为属于颜色的一种（RGBA4种），它是NGUI中继承了Widget的控件类才有的属性，如 Sprite、Texture、Label。如果父物体身上没有任何组件继承 Widget，则改变透明度不会影响到子物体。在这种情况下，如果我们又不能故意为父物体赋予一

个Sprite、Texture、Label等组件，可以给父物体增加一个Widget组件，增加方式为：在Inspector面板上，依次单击AddComponent→NGUI→UI→NGUI Widget。然后父物体身上的透明度动画就会带着子物体一起变化了。

## 9.6 为什么动画播放一遍之后无法再次正常播放

Tween动画播放如果是Once类型的，那么它播放一遍之后，这个动画就永久结束了，它的信息就永远是终点信息了。不管怎样去关闭再激活这个动画组件，它都无法再次进行播放。

如果要多次播放动画，可以实现将动画组件关闭，然后在需要播放的地方使用代码去对它进行手动播放。操作方法见7.6节。

我们也许还需要对动画进行复位操作，复位操作也可以在7.6节中看到。

在项目开发中如果要使用Tween动画，如果这个动画是Once类型并且需要多次调用，那么最好是事先将动画组件关闭禁用，然后手动使用代码调用它的播放函数进行播放，这样可以更深度地控制动画。

## 9.7 为什么3DUI模式下，UI资源的尺寸Snap后和屏幕的大小比例不一致

有时候有这么一种情况：美术切出一个资源文件，它的大小按理说应该占屏幕十分之一的高度，但是放到NGUI中使用时，发现使用



Snap还原它的原像素尺寸后，占屏幕的高度比例明显地比十分之一大或者小，这种情况有两种原因。

(1) 检查UIRoot的缩放类型和标准分辨率设置，看是否和美术人员作图使用的分辨率是一致的。一个游戏项目中一定会有一个标准参照分辨率。

(2) 在NGUI某些版本中的3DUI中，会明显地出现这个问题，在这种情况下，只需要将NGUI的Camera的Filed Of View设为75即可。

## 9.8 为什么UI不受灯光影响

有时候我们希望NGUI的控件能够受灯光影响，比如我们希望界面能反光。但是，无论怎么调整灯光的位置和Layer，都无法影响到NGUI的控件，这是因为图片的Shader导致的，比如我们在制作图集的时候，生成的图集的材质球的默认Shader是不受光照影响的。

我们以Sprite精灵图片为例，如果希望它受光照影响，那么制作图集的时候，在生成图集的材质球（制作图集会生成3个文件，材质球是其中一个）之后，将图集的材质球的Shader更换为一个可以反光的Shader，如Reflective等。

## 9.9 为什么3D模型放到UIRoot下就变得看不见了

图片的单位是像素，而Unity是个3D引擎，在Unity的世界里单位是米，而不是像素。为了让图片能在Unity中按照像素级别进行显示，

就需要UIRoot这个组件来对整个UI进行放缩，原理就是将米转换为像素。

所以，当把一个长宽高一米的三维模型放到UIRoot下之后，希望它被一起渲染时，会发现放到UIRoot下之后它看不见了，因为它的长宽高一米被UIRoot变成了一个像素单位（可以理解为1米变成了1像素），所以看不见了。这个时候可以修改模型的Scale缩放比例来让它显示出足够的大小来。

## 9.10 为什么UI单击后无法播放音效

这种情况，检查4个问题。

- （1）播放声音的组件或者逻辑是否正确。
- （2）使用设备（如计算机或者手机）的声音是否已经打开。
- （3）场景中是否有一个物体具有AudioListner组件（也就是MainCamera身上那个组件），如果没有这个组件，场景中所有的声音都听不见。可以在任何一个物体身上添加AudioListner组件。
- （4）检查声音的多普勒效应。在使用3D声音的情况下，音源离AudioListner所在的物体越远，声音会越小，受多普勒效应的影响。

## 9.11 为什么Depth更大的图片反而被Depth小的图片遮住

在NGUI中，如果显示顺序与自己预想的不一样，那么检查以下两点。

(1) 这两个图片是否属于不同的Panel? 如果是, 那么不论这两个图片的Depth怎么设置, 具有更高Depth的Panel的子物体, 会永远遮挡更低Depth的Panel的子物体。

(2) 如果Panel的Depth和图片本身的Depth的关系都没有问题, 那么检查Panel的RenderQ的值。

普遍来说, 第一种情况占大多数。

## 9.12 怎样判断点中的东西是UI

有时候我们在玩游戏时, 单击行为除了操作UI以外, 还有一些其他的游戏逻辑, 例如, 单击地面时移动角色等。在这种情况下, 我们也许会碰到一些事件击穿的问题, 例如, 单击了一个按钮, 按钮的单击事件触发了, 但是角色也跟着一起移动了。

这种情况下, 我们也许需要在逻辑的某些地方判断一下单击的地方是否是UI所处的地方。例如, 我们可以在玩家单击屏幕移动角色时, 加入以下判断即可:

```
if (UICamera.IsOverUI)
{
    //此时鼠标光标正在UI上, 不能移动角色
}
else
{
    //鼠标光标不在UI上, 点中的是地面, 可以移动角色
}
```

或者使用UICamera.IsPressed(GameObject go)方法判断是否正点住了某一个物体。

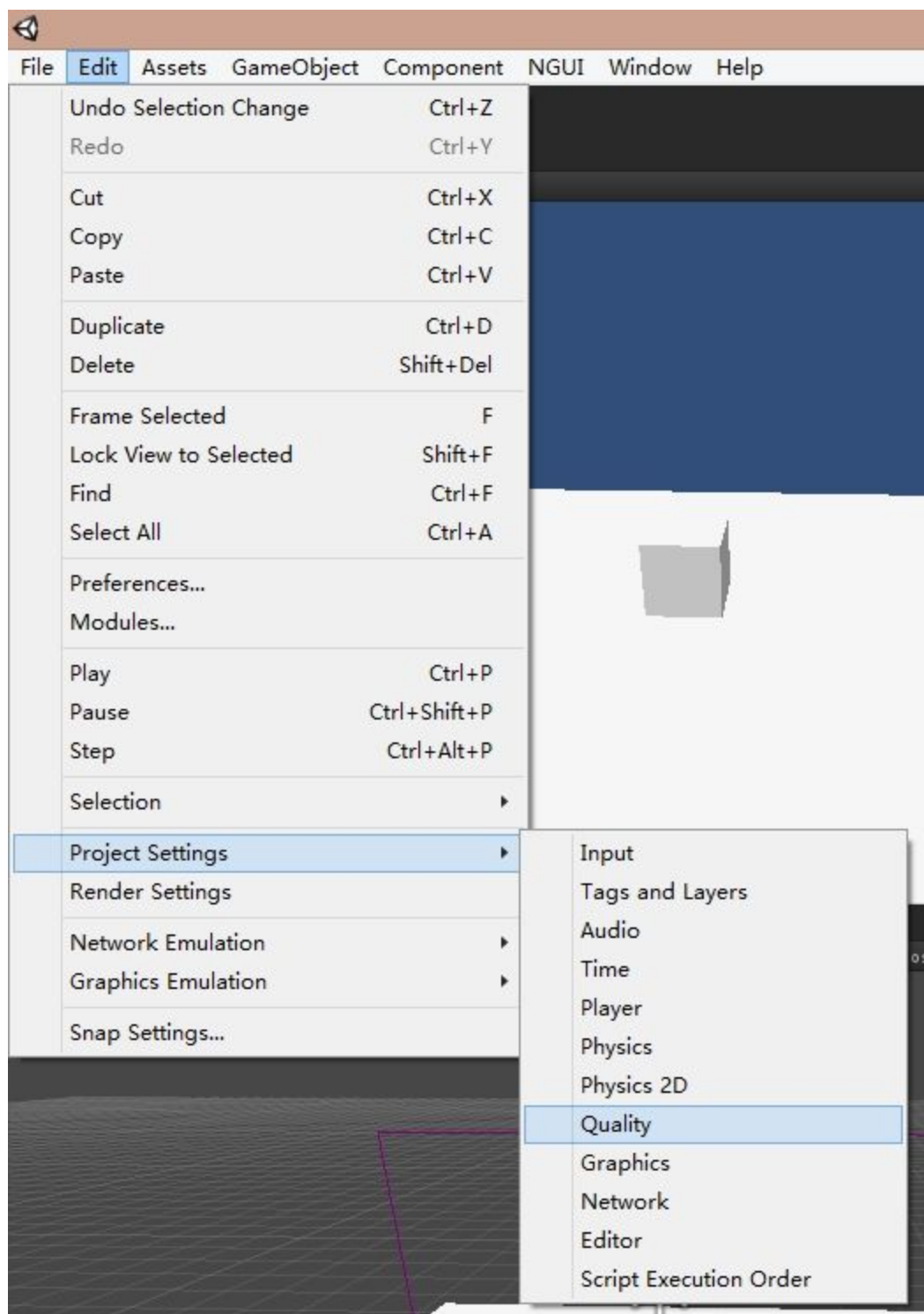
## 9.13 为什么Label的文字始终不够清晰、明亮

有时候我们使用Label时，发现文字不论调多大都不够清晰、明亮，有以下可能性。

（1）使用的静态字体。我们知道静态字体是一张预先制作好的图片，调用静态字体就像调用Sprite一样，所以当字体太大时，就相当于把图片放大了。自然会模糊。改成动态字体或者重做分辨率更高的静态字体可以获得更好的效果。

（2）一般来说NGUI中创建出来的新Label默认是勾选上了Gradient渐变的，并且默认的渐变色是从上往下、从白到灰色渐变，把这个勾选取消，字体也会显得更明亮。

（3）如果项目质量（Quality设置）过低也可能导致UI模糊，不仅是Label模糊。我们可以在Unity顶部Edit菜单中，选择ProjectSetting，选择Quality设置（操作方法如图9.2所示），然后将会在Inspector面板中显示图9.3所示的质量设置面板，在其中将质量调高即可。



▲ 图9.2



▲ 图9.3

## 9.14 为什么创建的物体有BoxCollider却无法接收事件

我们都知道 NGUI接收用户的操作事件几乎全部都依赖BoxCollider，但是，如果创建的物体有 BoxCollider，却无法接收到用户的操作事件，例如单击、拖动等，可以从以下几个方面检查。

- (1) 检查这个物体的Layer是否被UICamera所在的相机渲染到。
- (2) 检查这个物体的Layer在UICamera组件中的EventMask中是否被屏蔽了。
- (3) 检查这个物体上层是否有另一个具有BoxCollider的控件给挡住了。
- (4) 检查我们的事件监听逻辑是否写得正确。

## 9.15 为什么改变了控件的父物体，导致了显示层级错乱

这是一个比较令人头疼的问题。例如，有两个Panel，分别为PA和PB，其中 PA的Depth是1、PB的Depth是2。有一个控件X，它是PA的子物体，深度为10；有另一个控件Y，它是PB的子物体，深度为8。

在正常情况下，因为PB的深度大于PA的深度，所以，即使Y的深度低于X的深度，依靠Y所在的Panel的深度更高，Y会显示在X上层。

这个时候，如果在游戏运行过程中，使用代码动态地去改变X的父物体，例如：

```
X.transform.parent = PB.transform;
```

这个时候，X确实处于PB下面作为子物体了，而且Depth也依然是10，问题来了：此时X和Y都处于同一个Panel下，并且X的Depth为10高于Y的Depth=8，但是X很有可能依然像原来一样显示在Y的下层。

这种情况是因为在代码动态修改之后，NGUI没有刷新渲染信息导致的，要解决这个问题，可以通过手动刷新解决：

```
X.GetComponent<UISprite>().RemoveFromPanel();
```

```
X.transform.parent = PB.transform;
```

```
PB.GetComponent<UIPanel>().RebuildAllDrawCalls();
```

这样，就相当于手动将X从原来的Panel上移除，为它赋予新的Panel父物体，然后刷新这个新的Panel组件，让它重新构建一次DrawCall。

## 9.16 关于ScrollView滑动的问题

ScrollView的用法因为涉及多个组件，所以，也是经常遇到问题的地方，下面就来看一下ScrollView的一些常见问题的检查。

### 1. 滑动不了ScrollView了

如果出现这种情况，从以下几点开始检查。



- ScrollView** 下包含的内容是否有 **BoxCollider** 并能够正常接收事件（如单击等）。如果没有 **BoxCollider** 或者 **BoxCollider** 不能正常接收事件（如物体的 **Layer** 不对、被其他的**BoxCollider**遮挡等），则无法拖动**ScrollView**下的内容来滚动浏览。

- 检查**ScrollView**下包含的内容是否有**DragScrollView**组件，**ScrollView**要正常工作，必须要内容物体上有这个组件，这个组件和**BoxCollider**缺一不可。

## 2. **ScrollView**的滑动效果不是自己想要的

比如我们滑动时，希望它不要弹一下，或者到合适的程度就不能滑动了等，可以在**Scroll View**组件中进行设置。具体参看第3章中对**ScrollView**的讲解。